Lint-työkalun käyttöönoton vaikutus JavaScript-sovelluksen ylläpidettävyyteen

Sonja Somero

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 3. tammikuuta 2018

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author

Sonja Somero

Työn nimi — Arbetets titel — Title

Lint-työkalun käyttöönoton vaikutus JavaScript-sovelluksen ylläpidettävyyteen

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	3. tammikuuta 2018	50 sivua + 10 sivua liitteissä

Tiivistelmä — Referat — Abstract

Sovelluksen ylläpitäminen on usein kallista. Muutosten tekeminen ja vikojen korjaaminen on edullisempaa kehitysvaiheessa kuin ylläpitovaiheessa. Jotta mahdollisimman moni virhe löytyisi aikaisessa vaiheessa, ohjelmistoprojekteissa voidaan käyttää muun muassa staattisen analyysin työkaluja, joilla virheitä voidaan havaita. JavaScript on yksi käytetyimmistä ohjelmointikielistä, mutta se sisältää kuitenkin joukon huonoja käytänteitä. Lisäksi jokin asia, kuten funktion määrittely, voidaan toteuttaa usealla eri tavalla. Huonojen käytänteiden välttämiseksi ja ohjelmakoodin yhdenmukaisuuden lisäämiseksi JavaScriptille on kehitetty lint-työkaluja, jotka tekevät ohjelmakoodille staattista analyysia etsien edellä mainittujen ongelmien lisäksi syntaksivirheitä ja muita mahdollisia virheitä.

Tässä tutkielmassa tarkastellaan miten lint-tyyppisen ESLint-työkalun käyttöönottaminen vaikuttaa JavaScript-sovelluksen ylläpidettävyyteen. Tutkimus tehdään yksittäiselle projektille, jossa toteutettiin JavaScript-sovellus. Ylläpidettävyyttä verrataan metriikoiden avulla ennen ESLintin havaitseminen ongelmien korjaamista sekä korjausten jälkeen. Lisäksi kohdeprojektin kehittäjä arvioi ESLintin avulla korjattua ohjelmakoodia.

Tulosten perusteella ESLintin käyttöönotto on helppoa ja nopeaa ja sen käyttämisen katsotaan parantavan ylläpidettävyyttä. Lisäksi ESLintin käyttämisestä seurasi, että aiemmin piilossa olleet toisteiset rakenteet tulivat ohjelmakoodin yhdenmukaistumisen myötä esille.

ACM Computing Classification System (CCS):

Software and its engineering~Maintaining software

Software and its engineering~Software verification and validation

General and reference~Metrics

Avainsanat — Nyckelord — Keywords

ylläpidettävyys, staattinen analyysi, metriikat

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 Ylläpidettävyys **3**

2.1 Sovelluksen ylläpitäminen 3

2.2 Tekninen velka 5

2.3 Koodihajut 6

2.4 Ylläpidettävyyden mittaaminen 7

2.4.1 Mittareita 8

2.4.2 Mittareiden luotettavuus 12

3 Koodin analysointi **14**

3.1 Staattinen analyysi 14

3.2 Dynaaminen analyysi 18

3.3 JavaScriptin ongelmat 19

3.4 Työkalut JavaScriptille 23

3.4.1 ESLint 23

3.4.2 Muut lint-työkalut 26

4 Tutkimusasetelma **26**

4.1 Kohdeprojekti 27

4.2 ESLintin konfigurointi 28

4.3 Ylläpidettävyyden mittaaminen 29

5 Tulokset **30**

5.1 ESLintin havaitsemat ongelmat 30

5.2 Ylläpidettävyysmittausten tulokset 32

5.3 Kehittäjän näkemys ESLintin käyttöönotosta 34

6 Keskustelu **35**

6.1 ESLintin tulosten arviointi 35

6.2 ESLintin käyttöönotto ohjelmistoprojektissa 39
6.3 Rajoitteet ja mahdolliset jatkotutkimuskohteet 40
6.4 Tutkimuskysymyksiin vastaaminen 40

7 Yhteenveto **41**

Lähteet **43**

A ESLintin konfiguraatio **51**

B SonarQuben konfiguraatio **52**

C Täydellinen listaus ESLintin raportoimista virheistä **53**

1 Johdanto

JavaScript on viime vuosina noussut yhdeksi suosituimmista ja käytetyimmistä ohjelmointikielistä [SO16]. JavaScript on dynaamisesti tyypitetty kieli, joka kehitettiin ja julkaistiin hyvin nopeasti, ja jonka alkuperäisenä tarkoituksena oli vain hoitaa pieniä tehtäviä WWW-sivuilla [Cro08]. Nopean aikataulun vuoksi kieltä ei juurikaan testattu ennen julkaisua, ja myöhemmin JavaScriptin käytön laajentuessa sitä onkin kritisoitu siitä, että se sisältää paljon huonoja käytänteitä, joita ohjelmoijan kannattaisi välttää [Cro08].

JavaScriptin kehitys siirrettiin standardointiorganisaatio ECMA Internationalille, joka teki JavaScriptistä spesifikaation [Wik17]. ECMAScript-spesifikaatio [EA+97] on standardoitu ja siitä on julkaistu kahdeksan versiota. Uusin versio, ECMAScript 2017 [Int17], julkaistiin kesäkuussa 2017, mutta kaikkiin JavaScriptiä suorittaviin JavaScript-moottoreihin ei vielä ole implementoitu uusinta ECMAScript-versiota. ECMAScript 2015 [Int15], joka tunnetaan myös nimillä ECMAScript6 ja ES6, ja johon viitataan myös tämän tutkielman kokeellisessa osuudessa, on jo lähes täysin tuettu eri JavaScript-moottoreilla [Wik17].

JavaScriptin huonojen käytänteiden välttämiseksi on kehitetty erilaisia lint-työkaluja, joiden avulla ohjelmakoodista voidaan etsiä niin syntaksivirheitä kuin myös mahdollisia ongelmakohtia. Lint-työkalut ovat staattisen analyysin työkaluja, eli ohjelmakoodia ei suoriteta analysoinnin aikana. Tällaisia työkaluja ovat muun muassa ESLint [ESL17], JSLint [JSL17] ja JSHint [JSH17].

Kun sovellusta kehitetään eteenpäin, koodia voi olla hankalaa pitää yhdenmukaisena. Yhdenmukainen koodi on sellaista, jossa samantyyppiset asiat tehdään kaikkialla koodissa samalla tavalla [Mar08], eli ohjelmointityyli on yhdenmukaista. Esimerkiksi JavaScriptissä funktiot voidaan määritellä monella eri tavalla, joita on esitetty koodiesimerkissä 1.1.

Yhdenmukaisessa koodissa funktiot on määritellään kaikkialla samalla

syntaksilla. Jos ohjelmointityylin suhteen ei ole rajoituksia, voi sovelluksen ylläpitäminen tai uuden kehitystiimin ottaminen projektiin olla haasteellista JavaScriptin useiden eri toteutusmahdollisuuksien vuoksi. Tällöin pahimmassa tapauksessa joitakin osia sovelluksesta voidaan joutua kirjoittamaan uudestaan, jotta ohjelmakoodi olisi ymmärrettävää. Tässäkin tapauksessa jokin automaattinen lint-työkalu voi olla hyödyksi, sillä se ohjaa ohjelmoijaa pitämään koodin yhdenmukaisena. Kun koodi on yhdenmukaista, sitä on myös helpompi ylläpitää.

```
function name() {};
var name = function namedFunction() {};
var name = () => {};
```

Koodiesimerkki 1.1: Eri tapoja määritellä funktio JavaScriptillä.

Sovelluksen ylläpitäminen tarkoittaa jotakin prosessia, jossa sovellusta muutetaan käyttöönoton jälkeen, joko vian korjaamiseksi, toiminnallisuuden lisäämiseksi tai uuteen ympäristöön mukauttamiseksi [Som10]. Ylläpidettävyys kuvaa sitä, kuinka helposti sovellukseen pystytään tekemään muutoksia [ISO10]. Ylläpidettävyyteen vaikuttaa muun muassa teknisen velan määrä [TAV13]. Tekninen velka voi olla esimerkiksi ohjelmakoodia, joka toimii, mutta joka on toteutettu mahdollisimman nopeasti välittämättä laadusta. Ylläpidettävyyttä voidaan mitata esimerkiksi erilaisilla lähdekoodista laskettavilla metriikoilla [RMT09].

Tutkielman tavoitteena on selvittää, onko JavaScriptille kehitettyjen lint-työkalujen käyttöönotolla ja jonkin työkalun ohjeistuksen mukaan tehtävillä refaktoroinneilla vaikutusta sovelluksen ylläpidettävyyteen. Tutkielman kokeellisessa osuudessa lint-työkalun käyttöönoton vaikutuksia mitataan JavaScript-projektille. Projektille tehdään laatuanalyysia kahdessa tilanteessa: kun työkalu on otettu käyttöön mutta korjauksia ei ole vielä tehty sekä sen jälkeen, kun työkalun raportoimat ongelmat on korjattu. Laatuanalyysis-

sa keskitytään ylläpidettävyyttä kuvaaviin metriikoihin, joiden muutoksia verrataan edellä mainituissa kahdessa tilanteessa. Esimerkkityökaluna käytetään ESLintiä. Lisäksi projektissa mukana ollut kehittäjä arvioi muutoksia ja muutosten vaikutusta ylläpidettävyyteen kehittäjän näkökulmasta.

Tutkielman tutkimuskysymykset ovat seuraavat:

TK1: *Miten ESLintin käyttäminen vaikuttaa sovelluksen ylläpidettävyyteen?*

TK2: *Onko ESLintin käyttämisestä hyötyä ohjelmistokehityksessä?*

TK3: *Miten ESLint saadaan osaksi kehitysprosessia?*

Tutkielman rakenne on seuraava. Luvussa kaksi käsitellään ylläpidettävyyttä, siihen vaikuttavia tekijöitä ja sitä miten ylläpidettävyyttä voidaan mitata. Luvussa kolme perehdytään siihen, miten ohjelmakoodia voidaan analysoida, sekä käydään läpi JavaScriptin ongelmia ja esitellään analysointityökaluja, joilla näitä ongelmia voidaan ehkäistä. Neljännessä luvussa käydään läpi tutkimusasetelma ja viidennessä luvussa esitetään tutkimustulokset. Luvussa kuusi analysoidaan tuloksia, tehdään johtopäätökset sekä pohditaan mahdollisia rajoitteita ja jatkotutkimuskohteita. Seitsemännessä luvussa on tutkielman yhteenveto.

2 Ylläpidettävyys

Sovelluksen ylläpidettävyyteen vaikuttaa monet eri tekijät. Tässä luvussa perehdytään siihen, mitä sovelluksen ylläpitäminen tarkoittaa, mitkä tekijät vaikuttavat ylläpidettävyyteen ja miten ylläpidettävyyttä voidaan mitata.

2.1 Sovelluksen ylläpitäminen

Sovellusta ylläpidettäessä muutokset tehdään joko muokkaamalla sovelluksen vanhoja komponentteja tai lisäämällä siihen uusia. Ylläpitäminen voidaan

jakaa kolmeen eri tyyppiin: vikojen korjaamiseen (fault repairs), ympäristöön mukauttamiseen (environmental adaptation) ja toiminnallisuuden lisäämiseen (functional addition) [Som10].

Vikojen korjaaminen tarkoittaa minkä tahansa virheen korjaamista. Vikoja voivat olla muun muassa koodausvirheet, suunnitteluvirheet sekä virheet vaatimusmäärittelyssä. Ympäristöön mukauttamisella tarkoitetaan niitä muutoksia, jotka täytyy tehdä, jos esimerkiksi laitteistotasolla tai sovellusta tukevissa ohjelmistoissa tapahtuu muutoksia, joihin itse sovelluksen on mukauduttava toimiakseen. Toiminnallisuuden lisääminen voi tulla tarpeeseen, jos sovelluksen vaatimukset muuttuvat organisatoristen tai liiketoimintaan liittyvien muutosten myötä [Som10].

ISO/IEC on määritellyt ylläpidettävyyden kuvaamaan tasoa, jolla aiotut ylläpitäjät pystyvät tekemään tuotteeseen tai järjestelmään muutoksia tehokkaasti ja tuottavasti [ISO11]. Ylläpidettävyys siis kuvaa sitä, kuinka helposti sovellukseen pystytään tekemään muutoksia [ISO10]. Ylläpidettävyys on yksi ISO/IEC:n sovelluksen laatua arvioivan tuotelaatumallin kahdeksasta laatupiirteestä. Muut laatupiirteet ovat toiminnallinen sopivuus, tehokkuus, yhteensopivuus, käytettävyys, luotettavuus, turvallisuus ja siirrettävyys. Tuotelaatumallissa ylläpidettävyys jakautuu viiteen alipiirteeseen: modulaarisuuteen, uudelleenkäytettävyyteen, analysoitavuuteen, muunneltavuuteen ja testattavuuteen [ISO11].

Sovelluksessa piilevän ongelman löytäminen ja korjaaminen vasta sovelluksen toimittamisen jälkeen tulee usein kalliimmaksi kuin korjaaminen suunnittelu- ja kehitysvaiheessa [BB01]. Sovelluksen ylläpitäminen onkin usein kalliimpaa kuin itse kehittäminen. Ylläpitokuluihin voivat vaikuttaa monet eri tekijät, kuten sovelluksen ikä, koko ja uudelleenkehittäminen [Gui83]. Esimerkiksi mitä pidempään sovellus on ollut käytössä ja mitä enemmän siihen on ajan mittaan tehty muutoksia, sitä enemmän ylläpitokulut useimmiten kasvavat [Gui83].

2.2 Tekninen velka

Vuonna 1992 Ward Cunningham antoi ensimmäisen määritelmän tekniselle velalle [Cun92]. Määritelmän mukaan tekninen velka tarkoittaa koodia, joka toimii, mutta ei ehkä ole riittävän hyvin toteutettu ("not-quite-right code") [Cun92]. Yleensä tekninen velka tarkoittaakin kompromissia nopeamman julkaisemisen sekä laadun ja ylläpidettävyyden välillä. Toisin sanoen, jos jokin toiminnallisuus halutaan saada nopeammin valmiiksi, saatetaan koodin laadusta ja ylläpidettävyydestä joutua tinkimään [TAV13].

Teknistä velkaa on useaa eri tyyppiä, kuten koodivelkaa, arkkitehtuurivelkaa, suunnitteluvelkaa ja testivelkaa [LAL15]. Tässä tutkielmassa keskitytään teknisen velan osalta koodivelkaan. Koodivelka viittaa sellaisiin ongelmiin lähdekoodissa, jotka hankaloittavat koodin ylläpitoa sekä vaikeuttavat uuden koodin tuottamista [AMDM+16].

Teknistä velkaa voidaan ottaa sekä sovelluksen kehitysvaiheessa että ylläpitovaiheessa [TAV13], ja se voi olla joko tahallista tai tahatonta [YHMS15]. Tahallinen velka kehitysvaiheessa voi olla hyödyksi, jos velkaa ottamalla saadaan toteutettua nopeammin jokin toiminnallisuus, joka tuo liiketoiminnallista lisäarvoa [LAL15]. Tällaisessa tapauksessa on kuitenkin tärkeää, että velkaa hallitaan hyvin ja sen hinta pidetään näkyvillä [LAL15]. Tahaton tekninen velka voi olla hyvin haitallista, sillä kehittäjät eivät tiedä tahattoman velan sijaintia tai velan seurauksia [LAL15].

Teknisen velan määritelmä ei ole täysin yksiselitteinen, ja on eriäviä mielipiteitä siitä, mikä kaikki lasketaan tekniseksi velaksi. Osa mieltää sekä tahallisen että tahattoman velan tekniseksi velaksi, kun osa taas mieltää tekniseksi velaksi ainoastaan tahallisen velan [YHMS15]. Myös esimerkiksi legacy-järjestelmien vanhentuneet teknologiat voitaisiin joissain tapauksissa laskea tekniseksi velaksi [YHMS15]. Lisäksi uusien velkatyyppien lisääntyessä teknisen velan määritelmä todennäköisesti elää jatkuvasti.

2.3 Koodihajut

Koodihaju (code smell) on Martin Fowlerin ja Kent Beckin kehittämä termi, joka viittaa sellaiseen lähdekoodin kohtaan, joka jo nopealla vilkaisulla vaikuttaa merkiltä mahdollisesta ongelmasta [Fow06]. Yksi esimerkki koodihajusta on pitkä metodi.

Vaikka kaikki koodihajut eivät välttämättä osoita todellisiin ongelmiin koodissa [Fow06], koodihajuja pidetään usein merkkinä siitä, että ohjelmakoodia tulisi refaktoroida [FB99]. Refaktorointi tarkoittaa prosessia, jossa ohjelmaa muutetaan siten, että muutokset eivät vaikuta koodin ulkoiseen toimintaan, parantaen kuitenkin koodin sisäistä rakennetta [FB99]. Koodihajut saattavat hieman lisätä muutosten tekemisen työläyttä ja huonontaa sovelluksen ylläpidettävyyttä, mutta merkittäviä eroja ei kuitenkaan ole vielä havaittu [SYA$^+$13].

Eräs koodihaju on "perinnästä kieltäytyminen" (refused bequest), joka tarkoittaa, että aliluokka ei halua tai sen ei tarvitse käyttää kaikkia ominaisuuksia, jotka se perii, ja näin ollen luokkahierarkia saattaa olla virheellinen [FB99]. Vaikka yleensä koodihajujen löytyminen ohjelmakoodista mielletään negatiivisena asiana, tämän koodihajun esiintymisen on havaittu vaikuttavan myönteisesti sovelluksen ylläpidettävyyteen, ja muutosten tekeminen on ollut vähemmän työlästä [SYA$^+$13].

Vuonna 2011 tehdyssä systemaattisessa kirjallisuuskatsauksessa selvisi, että aiemmat koodihajuihin liittyvät tutkimukset ovat enimmäkseen keskittyneet metodeihin ja työkaluihin, joilla koodihajuja voidaan löytää [ZHB11]. Puolestaan sen tutkiminen, viittaavatko koodihajut ongelmiin lähdekoodissa, on jäänyt vähemmälle. Muutamissa tutkimuksissa on tutkittu viiden eri koodihajun vaikutuksia Fowlerin ja Beckin [FB99] määrittelemästä 22:sta koodihajusta. Esimerkiksi koodin toisteisuuden on havaittu huonontavan ylläpidettävyyttä, mutta toisaalta joissakin tapauksissa toisteisuus voi olla hyvästä parantaen luotettavuutta [ZHB11]. Isoilla luokilla, pitkillä metodeil-

la ja haulikkokirurgialla[1] (shotgun surgery) puolestaan on havaittu olevan tilastollinen yhteys ohjelmistovikoihin [ZHB11]. Toisaalta isojen luokkien refaktorointi voi vaikeuttaa sovelluksen myöhempää ylläpitoa, sillä aiemmin yhdessä isossa luokassa ollut toiminnallisuus on pilkottu useampaan osaan, jolloin kaiken kaikkiaan järjestelmän koko kasvaa [SAM12]. Lisäksi on havaittu, että sellaiset ohjelman luokat, joissa on koodihajuja, ovat todennäköisemmin alttiimpia muutoksille kuin sellaiset luokat, joissa koodihajuja ei ole [KPG09].

Yhteenvetona voidaan sanoa, että osalla koodihajuista saattaa olla negatiivinen vaikutus sovelluksen ylläpidettävyyteen, mutta jotkin hajut voivat toisaalta helpottaa muutosten tekemistä ohjelmakoodiin. Tutkimuksissa ei kuitenkaan ole ilmennyt selvää näyttöä sen suhteen, että koodihajut huonontaisivat ylläpidettävyyttä merkittävästi [SYA+13, ZHB11]. Koodihajujen vähentämistä tärkeämpää vaikuttaisi olevan ohjelmakoodin määrän vähentäminen ja se, ettei koodin määrä kokoajan kasvaisi [SYA+13]. Koodihajujen vaikutusta ohjelmakoodiin ja sen myötä ylläpidettävyyteen on vielä tarpeen tutkia lisää luotettavien tulosten aikaansaamiseksi.

2.4 Ylläpidettävyyden mittaaminen

Ohjelmistotuotannossa käytettävät mittarit voivat olla joko suoria (direct) tai johdettuja (indirect, surrogate). Suora mittari ei ole riippuvainen minkään muun ominaisuuden mittauksesta. Johdettu mittari puolestaan saadaan yhden tai useamman muun ominaisuuden mittauksesta [ISO10]. Tässä aliluvussa käsitellään mittareita, joiden avulla voidaan kuvata sovelluksen ylläpidettävyyttä.

[1] Haulikkokirurgialla tarkoitetaan sitä, kun jonkin pienen muutoksen tekeminen vaatii muutoksia useaan eri kohtaan.

2.4.1 Mittareita

Ylläpidettävyyden mittaamiseen ja ennustamiseen käytetään useimmiten sellaisia lähdekoodista laskettavia mittareita, jotka mittaavat sovelluksen kokoa, kompleksisuutta tai kytkentää [RMT09]. Pelkän yhden mittarin käyttäminen ylläpidettävyyden tai ylipäätään sovelluksen laadun mittaamisessa ja ennustamisessa ei välttämättä takaa luotettavia tuloksia, vaan ylläpidettävyyden arvioimisessa niitä on parempi käyttää useampia [Mis05, WH88].

Ylläpidettävyyden mittaamiseen käytettäviä mittareita on listattuna taulukossa 2.1. Niin sanotut perusmittarit, kuten koodirivien määrä (number of lines of code, LOC) ja kommenttirivien määrä (number of lines of comments, CMT) vaikuttavat merkittävästi ylläpidettävyyteen: mitä enemmän koodirivejä tai vähemmän kommenttirivejä sovelluksessa on, sitä vaikeampi sitä on ylläpitää [WOA97]. Suuri kommenttirivien määrä ei kuitenkaan suoraan indikoi hyvää ylläpidettävyyttä, sillä kommenttien sisältö vaikuttaa asiaan. Kommentit voivat esimerkiksi olla vanhentunutta tietoa sisältäviä tai muuten harhaanjohtavia. Lisäksi suurta kommenttien määrää parempana vaihtoehtona pidetään useimmiten sitä, että koodi selittää itse itsensä, eli se on helppolukuista ja ymmärrettävää myös ilman kommentteja [Mar08].

Kuten koodirivien määrä, myös metodin ja luokan koko vaikuttavat sovelluksen ylläpidettävyyteen. Mitä suurempi luokka tai metodi on, sitä monimutkaisempi ja vaikeammin ymmärrettävä se on [Mis05]. Kytkentä (coupling) tarkoittaa sovelluksen moduulien välistä riippuvuussuhdetta [ISO10]. Vahva kytkentä sovelluksen moduulien välillä tekee sovelluksesta monimutkaisemman ja vaikeammin ylläpidettävän [Mis05]. Koheesiolla (cohesion) tarkoitetaan tasoa, jolla yksittäisen sovelluksen moduulin tehtävät tai metodit liittyvät toisiinsa [ISO10], ja sitä pidetään kytkennän vastakohtana. Matala koheesio viittaa vaikeammin ylläpidettävään sovellukseen: jos sovelluksen koheesio on matala, eli esimerkiksi metodit eivät liity toisiinsa luokkien sisällä, sovellus on monimutkaisempi kuin sellainen sovellus, jolla on korkea

koheesio [Mis05]. Näiden lisäksi ylläpidettävyyttä voidaan mitata myös joidenkin monimutkaisempien mittarien avulla, jotka esitellään seuraavaksi.

Mittari	**Mittarin arvon muutoksen vaikutus ylläpidettävyyteen**
Koodirivien määrä	Koodirivien lisääntyminen huonontaa ylläpidettävyyttä.
Kommenttirivien määrä	Kommenttirivien lisääntyminen parantaa ylläpidettävyyttä.
Metodin tai luokan koko	Metodin tai luokan koon kasvaminen huonontaa ylläpidettävyyttä.
Kytkentä	Vahvan kytkennän omaavaa ohjelmistoa pidetään hankalammin ylläpidettävänä.
Koheesio	Korkean koheesion omaavaa ohjelmistoa pidetään helpommin ylläpidettävänä.
Syklomaattinen ja kognitiivinen kompleksisuus	Kompleksisuuden kasvaminen vaikeuttaa ylläpdiettävyyttä.
Ohjelman koko (Halstead)	Suurempi ohjelman koko vaikeuttaa ylläpidettävyyttä.
Ohjelmointiin kuluva aika (Halstead)	Mitä enemmän ohjelmointiin kuluu aikaa, sitä huonompi ylläpidettävyys on.
Ylläpidettävyysindeksi	Indeksin arvon kasvaminen kuvaa ylläpidettävyyden paranemista.

Taulukko 2.1: Ylläpidettävyyden mittaamiseen käytettäviä mittareita.

Syklomaattinen kompleksisuus

McCaben syklomaattinen kompleksisuus (cyclomatic complexity) kuvaa ohjelman monimutkaisuutta. Kompleksisuusarvo ohjelmalle saadaan laskemalla pitkittäissuunnassa riippumattomien polkujen enimmäismäärä [McC76]. McCaben syklomaattinen kompleksisuus ei kuitenkaan ota huomioon, miten polun haaraumien määrä – ja näin ollen kompleksisuusarvo – lasketaan, jos esimerkiksi if-lause sisältää useamman ehdon [Mye77].

Ongelman ratkaisemiseksi Myers kehitti laajennetun syklomaattisen kompleksisuuden (extended cyclomatic complexity). Laajennoksessa komplek-

sisuudelle lasketaan väli, jossa kompleksisuuden alaraja on haaraumien lukumäärä, johon lisätään luku yksi, ja yläraja on yksittäisten ehtojen lukumäärä, johon lisätään luku yksi [Mye77].

Myöhempien tutkimusten mukaan sekä McCaben syklomaattinen kompleksisuus että Myersin laajennettu syklomaattinen kompleksisuus vaikuttaisivat kuitenkin olevan yhtä hyviä ohjelman ylläpidettävyyden ennustamisessa [WOA97].

Kognitiivinen kompleksisuus

Syklomaattisen kompleksisuuden rinnalle on kehitetty toinen kompleksisuutta kuvaava mittari, kognitiivinen kompleksisuus (cognitive complexity) [SW03]. Kognitiivinen kompleksisuus mittaa sitä, kuinka kuormittavaa ohjelmakoodin jonkin osan ymmärtäminen on. Toisin sanoen se mittaa ohjelmakoodin osan vaikeutta tai suhteellista aikaa ja vaivaa, joka vaaditaan sovelluksen osan ymmärtämiseksi. Kognitiivinen kompleksisuus ottaa huomioon sekä sovelluksen sisäiset rakenteet että prosessoitavat syötteet ja tulosteet (I/O), toisin kuin syklomaattinen kompleksisuus joka ei huomioi I/O:ta [SW03].

Halsteadin metriikat

Halsteadin metriikat on kehitetty mittaamaan ohjelman eri ominaisuuksia. Halsteadin metriikoita ovat ohjelman pituus (program length), ohjelman koko (program volume), hankaluus ymmärtää tai kirjoittaa ohjelmakoodia (program difficulty) ja ohjelman kehittämiseen tai ylläpitämiseen vaadittava vaiva (program effort) [Hal77]. Ohjelman koko ja ohjelman kehittämiseen tai ylläpitämiseen vaadittava vaiva ennustavat hyvin ylläpidettävyyttä [WOA97]. Molemmat näistä lasketaan kaavoilla, joiden muuttujina ovat operaattorien ja operandien kokonaismäärä sekä selvästi erilaisten operaattorien ja operandien lukumäärä [Hal77]. Operaattoreita ovat muun muassa `+`, `<` sekä `==`.

Operandit puolestaan ovat operaattorien kohteita, eli esimerkiksi vakioita, muuttujia sekä metodien palauttamia arvoja.

Ylläpidettävyysindeksi

Syklomaattisen kompleksisuuden ja Halsteadin metriikoiden pohjalta on kehitetty ylläpidettävyysindeksi (maintainability index, MI). Ylläpidettävyysindeksi on yhdistelmämetriikka, joka mittaa sovelluksen ylläpidettävyyttä [WOA97]. Se on yksi eniten käytetyimmistä ylläpidettävyyden mittareista [SAM12], joka mittaa ylläpidettävyyttä lähdekoodin perusteella, eli kuvaa ohjelmakoodin ylläpidettävyyttä [WOA97]. Indeksin arvon kasvaessa sovelluksen ylläpidettävyys paranee [WOA97].

Useiden ohjelmakoodista laskettavien metriikoiden on havaittu olevan korrelaatiossa ylläpidettävyysindeksiin. Tällaisia metriikoita ovat muun muassa keskimääräinen luokan koodirivien määrä (average class size, ACLOC), keskimääräinen metodin koodirivien määrä (average method size, AMLOC), keskimääräinen metodin polun syvyys (average depth of paths, AVPATHS), ohjauskäskyjen osuus ohjelmakoodissa (control density, CDENS), kytkentä (coupling, COF), luokkahierarkian syvyys (depth of inheritance tree, DIT) ja uniikkien operaatioiden ja operandien määrä ohjelmassa (program vocabulary, n) [Mis05]. Kun näistä jonkun metriikan arvo muuttuu, myös ylläpidettävyysindeksin arvo muuttuu [Mis05].

Etenkin AMLOC:n on havaittu olevan hyvin paljon yhteydessä ylläpidettävyysindeksiin ja sitä pidetään yhtenä tärkeimmistä ylläpidettävyyden tekijöistä [Mis05]. Kun AMLOC:n arvo kasvaa, eli keskimääräinen metodin koko kasvaa, metodista tulee monimutkaisempi ja vaikeammin ymmärrettävä, jolloin myös ylläpitäminen vaikeutuu. Koska erilaisia ylläpidettävyyteen vaikuttavia tekijöitä on useita, ohjelmoijien olisikin hyvä tiedostaa ainakin AMLOC:n vaikutukset tehdessään sovellukseen muutoksia, jotta hyvä ylläpidettävyys säilyy [Mis05].

ISO/IEC:n laatumittarit

ISO/IEC on määritellyt ylläpidettävyydelle laatumittareita, joiden avulla voidaan kuvata sitä, kuinka tehokkaasti tuotteeseen voidaan tehdä muutoksia [ISO15]. Mittareilla mitataan ylläpidettävyyden alipiirteitä eli modulaarisuutta, uudelleenkäytettävyyttä, analysoitavuutta, muunneltavuutta ja testattavuutta. Nämä laatumittarit ovat komponenttien kytkentä, syklomaattinen kompleksisuus, moduulien uudelleenkäytettävyys, ohjelmointisääntöjen yhdenmukaisuus, järjestelmälokituksen täydellisyys, vianmääritysfunktioiden tehokkuus, vianmääritysfunktioiden riittävyys, muutosten tekemisen tehokkuus, muutosten virheettömyys, kyvykkyys tehdä muutoksia, testifunktioiden täydellisyys, testien riippumattomuus muista järjestelmistä ja testien uudelleenkäynnistettävyys.

Laatumittarit kuvataan kuitenkin hyvin yleisellä tasolla ja mittareiden arvot lasketaan suhdelukuina. Esimerkiksi syklomaattisen kompleksisuuden mittarissa lasketaan suhdeluku tietyn kompleksisuuden ylittävien komponenttien ja kaikkien tuotteen komponenttien välillä. Mittarit eivät kuitenkaan erikseen määrittele, millaisella kaavalla kompleksisuus lasketaan vaan se täytyy määritellä itse.

2.4.2 Mittareiden luotettavuus

Useimmat ohjelmistotuotannossa käytettävistä mittareista ovat johdettuja [KMB04]. Koska johdetut mittarit mittaavat asioita, joita ei voida mitata suoraan yhden muuttujan avulla [ISO10], tulisi johdetut mittarit aina validoida. Validoinnilla pyritään varmistumaan siitä, että mittarilla mitataan juuri sitä, mitä yritetäänkin mitata. On myös mittareita, jotka ensi alkuun kuulostavat suorilta mittareilta, mutta eivät sitä kuitenkaan välttämättä ole. Jopa koodirivien määrän oleminen suora mittari voidaan kyseenalaistaa pohtimalla, mitä ylipäätään tarkoittaa koodi tai rivi, ja miten näiden mieltäminen vaikuttaa rivien määrään tai siihen, mitä koodirivien

määrä ylipäätään tarkoittaa [KMB04].

Eräässä tutkimuksessa [SAM12] ylläpidettävyysindeksin, kytkennän ja luokkahierarkian syvyyden havaittiin olevan ristiriidassa ylläpidettävyyden kanssa. Koheesion käänteisluku ja sovelluksen koon mittarit vaikuttivat puolestaan ennustavan paremmin ylläpidettävyyttä ohjelmistoprojektin laajuudesta riippumatta. Mikäli monimutkaisempia mittareita halutaan käyttää ylläpidettävyyden mittaamiseen, on tärkeää varmistaa että käytettävät mittarit on arvioitu toimiviksi mitattavassa kontekstissa [SAM12].

Mittareiden arvioimiseksi on kehitetty malli, jonka avulla mittarin validiteetti ja riskit voidaan kartoittaa [KMB04]. Mallissa esitetään kymmenen kysymystä koskien arvioitavaa mittaria. Kysymykset on esitetty taulukossa 2.2.

Mikä on mittarin tarkoitus?
Mikä on mittarin laajuus?
Mitä attribuuttia yritetään mitata?
Mikä on mitattavan ominaisuuden luonnollinen asteikko?
Mikä on mitattavan ominaisuuden luonnollinen vaihtelevuus?
Mikä on käytettävä metriikka, eli funktio, joka määrittelee arvon mitattavalle ominaisuudelle? Mitä välinettä mittaukseen käytetään?
Mikä on metriikan luonnollinen asteikko?
Mikä on välineen (instrument) lukemien luonnollinen vaihtelevuus?
Mikä on atrribuutin suhde metriikan arvoon?
Mitkä ovat välineen käytön luonnolliset ja ennalta-arvattavat sivuvaikutukset?

Taulukko 2.2: Mittareiden arviointiin kehitetyn mallin [KMB04] kysymykset

Taulukon 2.2 kysymyksiin vastaamalla päästään paremmin perille arvioitavasta metriikasta – mittaako se todella sitä mitä halutaan mitata.

3 Koodin analysointi

Kun sovellusta kehitetään tai vanhaa ohjelmakoodia muokataan, on aina mahdollista, että ohjelmakoodia kirjoittaessa syntyy syntaksivirheitä, koodissa on muita vikoja tai koodin rakenne ei vastaa sovittuja käytänteitä. Jotta mahdollisimman moni vika löytyisi jo ohjelmointi- ja testausvaiheessa, on kehitetty erilaisia tekniikoita, joilla sovellusta voidaan analysoida. Tällaisia tekniikoita ovat staattinen ja dynaaminen analyysi, joita molempia varten on kehitetty lukuisia eri työkaluja.

Jos vikoja pystytään havaitsemaan jo kehitysvaiheessa, niiden korjaaminenkin on helpompaa ja halvempaa kuin vasta ylläpitovaiheessa [BB01, LSS07]. Tämän takia erilaisten staattisen ja dynaamisen analyysin työkalujen käyttäminen kehitysvaiheessa voi olla erittäin hyödyllistä. Mikään vikojen havaitsemistekniikka tai -työkalu ei kuitenkaan ole niin tarkka, että se pystyisi löytämään kaikki mahdolliset viat [YT89].

Tässä luvussa perehdytään tekniikoihin, joilla koodia voidaan analysoida, ja käydään läpi joitakin JavaScriptin tunnettuja ongelmia sekä apukeinoja näiden ongelmien havaitsemiseen koodia analysoimalla.

3.1 Staattinen analyysi

Staattinen analyysi analysoi ohjelmakoodia suorittamatta sitä [ISO10], ja sillä tarkoitetaan yleensä analyysia, joka suoritetaan jonkin työkalun avulla. Työkalun avulla tehtävää analyysia pidetään edullisena tapana laajentaa sovelluksen laadunvarmistusta [ZWN+06]. Kuitenkin myös ihmisen tekemä koodin katselmointi on staattista analyysia, ja staattisen analyysin työkalun käyttämisen katsotaan olevan ikään kuin esikatselmointia. Jonkin staattisen analyysin työkalun käyttäminen voi helpottaa varsinaista koodin katselmointia, sillä työkalun avulla ohjelmakoodista on mahdollista kitkeä pois muun muassa syntaksivirheitä, jolloin nämä eivät vie kaikkea huomiota katselmoinnissa [LSS07]. Katselmoinnin ideana on, että joku muu kuin koodin

tuottanut henkilö lukee tuotettua koodia. Katselmointi voi olla esimerkiksi tuotetun koodin muodollisempaa tai vapaamuotoisempaa läpikäyntiä koodin tuottaneen ohjelmoijan ja toisen henkilön kesken, mutta myös pariohjelmoinnin katsotaan kattavan katselmoinnin [HK07].

Syntaksivirheiden lisäksi staattisen analyysin työkaluilla ohjelmakoodista voidaan havaita muita poikkeavuuksia, jotka viittaavat mahdollisiin ohjelmointivirheisiin tai puutteisiin ohjelmakoodissa [Som10]. Poikkeavuudet voivat liittyä vikoihin datassa (data fault), kontrollissa (control fault), liittymäkohdissa (interface fault), muistin hallinnassa (storage management fault) tai syötteessä tai tulosteessa (input/output fault). Havaittavia vikoja ovat esimerkiksi määrittelemättömien muuttujien käyttö, käyttämättömien muuttujien määrittely, ohjelmakoodi jota ei koskaan saavuteta sekä funktion tulosten käyttämättä jättäminen. Eräässä tutkimuksessa suurin osa staattisen analyysin työkalun löytämistä vioista olivat sellaisia, jotka voisivat mahdollisesti haavoittaa sovelluksen turvallisuutta [ZWN+06].

Erilaisia staattisen analyysin työkaluja on lukuisia[2]. Niitä on sekä monia eri ohjelmointikieliä tukevia että jotakin tiettyä kieltä tukevia. Useita ohjelmointikieliä tukeva työkalu on esimerkiksi SonarQube [Son17], joka analysoi sovelluksen laatua mitaten muunmuassa kompleksisuutta, toisteisuutta, yksikkötestejä, haavoittuvuuksia ja ohjelmointistandardeja. Eri ohjelmointikielille on kehitetty erilaisia työkaluja, jotka keskittyvät esimerkiksi haavoittuvuuksien analysoimiseen [Che17a, Syn17] tai koodaustyylin analysoimiseen [Act17, Che17b, Mic17a]. Myös ohjelmointiympäristöt (integrated development environment, IDE) tekevät koodille staattista analyysia [Fou17a, Jet17, Mic17b].

Työkaluja, jotka etsivät ohjelmakoodista mahdollisia virheitä, kutsutaan usein lint-työkaluiksi. Nämä lint-työkalut ovat ohjelmakoodin oikeellisuuden tarkistustyökaluja, jotka tekevät koodille staattista analyysia. Ensimmäinen

[2] https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

lint-työkalu kehitettiin C-ohjelmointikielelle jo 1970-luvulla. Tämän työkalun tarkoitus on analysoida ja tarkistaa ohjelmakoodi ennen kuin se annetaan C:n kääntäjälle [Joh77].

Lint-työkalujen käyttöä on tutkittu aikaisemmin hieman. Esimerkiksi Android-pohjaisten mobiilisovellusten suorituskyvyn on havaittu parantuneen, kun kehityksen yhteydessä on käytetty Android Lint -työkalua mahdollisten ongelmien havaitsemisessa [NHN16]. Android Lint analysoi lähdekoodia etsien mahdollisia vikoja sekä kohtia, joita voisi optimoida toiminnan oikeellisuuden, turvallisuuden, suorituskyvyn, käytettävyyden, saatavuuden ja kansainvälistämisen kannalta.

Staattisen analyysin työkalut voivat myös tehdä vääriä havaintoja (false positive) [Som10]. Väärien havaintojen määrää on kuitenkin mahdollista vähentää asettamalla työkaluun kyseiselle ohjelmakoodille paremmin sopivia määrityksiä.

Koodin analysointi SonarQubella

SonarQube [Son17] tekee staattista analyysia annettujen sääntöjen perusteella. Se laskee sääntöjen perusteella erilaisia metriikoita, joiden avulla sovelluksen laadusta voidaan tehdä päätelmiä. Metriikat liittyvät kompleksisuuteen, dokumentaatioon, toisteisuuteen, ongelmiin, ylläpidettävyyteen, luotettavuuteen, turvallisuuteen ja testeihin.

SonarQubessa kompleksisuuden mittaamiseen käytetään kognitiivista kompleksisuutta. Kun funktion kontrollin kulku (control flow) katkeaa, kompleksisuus kasvaa yhdellä. Eri ohjelmointikielille on määritelty eri tapaukset, joissa kontrollin kulun katsotaan katkeavan, eli tapaukset, jotka kasvattavat kompleksisuutta. JavaScriptille SonarQubessa on määritelty seuraavat tapaukset kasvattamaan kompleksisuusarvoa: funktio, if-lause, &&-konjunktio, ||-konjunktio, ehdollinen kolmimuuttujalause, silmukka, `switch`-lauseen `case`-lausekkeet, `throw`- ja `catch`-lauseet sekä `return`-lause, mikäli

se ei ole funktion viimeinen lause.

Dokumentaatioon liittyvät metriikat käsittelevät lähdekoodissa olevia kommenttirivejä. Tällaisia metriikoita ovat kommenttirivien määrä sekä kommenttirivien tiheys prosentteina, jossa 50 % tarkoittaa että kommenttirivien ja koodirivien määrä on sama ja 0 % sitä, että koodi ei sisällä yhtään kommenttiriviä.

SonarQube etsii sovelluksesta toisteisia rakenteita. Toisteisuutta kuvataan toisteisuutta sisältävien rivien prosenttiosuudella sekä toisteisuutta sisältävien lohkojen, tiedostojen ja rivien lukumäärällä.

Analyysin aikana jokainen SonarQubeen määritellyn säännön rikkoutuminen tuottaa ongelman. Ongelmiin liittyvät metriikat kuvaavat muun muassa ongelmien määrää sekä uusien ongelmien määrää verrattuna edelliseen analyysiin.

Ylläpidettävyyteen liittyvät metriikat mittaavat koodihajuja, teknistä velkaa sekä ylläpidettävyystasoa. Koodihajuille lasketaan yhtenä metriikkana koodihajujen lukumäärä ja toisena uusien koodihajujen määrä verrattuna edelliseen analysointikertaan. Koodihajut määräytyvät SonarQubeen asetettavien sääntöjen perusteella. Säännöille määritellään tyyppi, joka voi olla joko virhe, haavoittuvuus tai koodihaju. Kun koodihaju-tyypin omaavaa sääntöä rikotaan, koodihajujen määrä kasvaa yhdellä.

Tekninen velka lasketaan sen perusteella, kuinka kauan kestäisi korjata kaikki ylläpidettävyyteen liittyvät ongelmat, eli koodihajut. Jokaiselle SonarQuben säännölle on määritelty sääntörikkomuksen korjaamiseen kuluva aika, jolloin tekninen velka on siis kaikkien koodihajujen korjaamiseen kuluva aika. Lisäksi SonarQube laskee teknisen velan suhteellisen osuuden (technical debt ratio), joka on sovelluksen korjaamisen hinnan (eli teknisen velan) ja sovelluksen kehittämisen hinnan välinen suhdeluku. Sovelluksen kehittämisen hinta on yhden koodirivin hinta kerrottuna koodirivien määrällä. Hinta on ajan yksikössä, ja yhden koodirivin toteuttamiseen kuluu oletusarvoisesti

30 minuuttia aikaa. Aika voidaan kuitenkin määritellä projektikohtaisesti SonarQubeen.

Ylläpidettävyystaso perustuu teknisen velan suhteelliseen osuuteen (technical debt ratio). Tason luokittelu on välillä A-E. A on paras taso, jolloin teknisen velan suhteellinen osuus on pieni, enintään 5 %. Tasolla E teknisen velan suhteellinen osuus on puolestaan yli 50 %.

Luotettavuutta mitataan virheiden määrällä. Sellaisten sääntöjen rikkominen, joiden tyyppi on "virhe", kasvattavat virheiden määrää. Lisäksi SonarQube laskee luotettavuustason välillä A-E, joka määräytyy virheiden lukumäärän mukaan. Tasolla A sovelluksessa ei ole yhtään virhettä ja tasolla E sovelluksessa on vähintään yksi häiriön aiheuttava virhe (blocker bug).

Turvallisuuden metriikoita ovat haavoittuvuuksien lukumäärä sekä turvallisuustaso. Sellaisten sääntöjen rikkominen, joiden tyyppi on "haavoittuvuus", kasvattavat haavoittuvuuksien määrää. Lisäksi SonarQube laskee turvallisuustason välillä A-E, joka määräytyy haavoittuvuuksien lukumäärän ja kriittisyyden mukaan. Tasolla A sovelluksessa ei ole yhtäkään havaittua haavoittuvuutta ja tasolla E sovelluksessa on vähintään yksi häiriön aiheuttava haavoittuvuus (blocker vulnerability).

Testeihin liittyvät metriikat mittaavat muun muassa ehtokattavuutta (condition coverage), rivikattavuutta, yksikkötestien määrää sekä kattamattomien rivien ja ehtojen määrää.

3.2 Dynaaminen analyysi

Dynaamisessa analyysissa ohjelmakoodia analysoidaan sen suorituksen aikana [ISO10]. Ehkä tunnetuin tapa suorittaa dynaamista analyysia on testaus. Siihen kuuluvat kaikki testauksen muodot, kuten yksikkötestaus, komponenttien testaus, järjestelmätestaus ja hyväksymistestaus [Som10].

Dynaamisen analyysin työkaluja voidaan hyödyntää esimerkiksi muistin käyttöön liittyvien vikojen etsimisessä. Purify on yksi esimerkki tällaises-

ta työkalusta. Se kehitettiin jo 1990-luvun alussa löytämään muistivuotoja ja pääsyvirheitä (access error) [HJ91]. Muistin käyttöön liittyvien tapausten lisäksi dynaamisen analyysin avulla on myös mahdollista seurata suorituskykyä [Cri17] sekä tarkkailla ohjelmakoodin toimintaa jäljittämällä, keskeyttämällä tai manipuloimalla ohjelmakoodin dataa suorituksen aikana [Mai17].

3.3 JavaScriptin ongelmat

JavaScript-sovellusten kehittäjät eivät tutkimusten mukaan ole aina tietoisia ECMA Internationalin julkaisemien JavaScriptin standardoitujen versioiden uusista ominaisuuksista tai siitä, miten jotakin uutta ominaisuutta tulisi käyttää [HHKWB16]. Uusien JavaScript-versioiden ominaisuuksien käyttöönottoa hankaloittaa myös se, että JavaScriptiä käyttävissä sovelluksissa on käytössä vanhoja ei-standardoituja sekä vanhentuneita JavaScriptin ominaisuuksia [HHKWB16].

Tämän lisäksi JavaScriptin toteutuksessa katsotaan olevan joitakin huonoja ominaisuuksia, joista ohjelmoijan on hyvä olla tietoinen ja joita kannattaa mahdollisesti välttää [Cro08]. Tunnetuimmat näistä niin kutsutuista JavaScriptin sudenkuopista esitellään seuraavissa aliluvuissa. Aliluvut pohjautuvat Douglas Crockfordin kirjaan *JavaScript: The Good Parts* [Cro08].

Globaalit muuttujat ja näkyvyysalue

Globaalit muuttujat ovat sellaisia muuttujia, jotka ovat näkyvillä kaikilla näkyvyysalueilla, eli joita mikä tahansa osa sovellusta voi käyttää tai muuttaa. Globaali muuttuja voidaan määritellä joko sijoittamalla `var`-lauseke funktion ulkopuolelle, lisäämällä ominaisuus (property) globaalille objektille tai käyttämällä muuttujaa määrittelemättä sitä (implied global). Globaalit muuttujat voivat hankaloittaa pienten itsenäisten aliohjelmien suoritusta, jos aliohjelmissa on saman nimisiä globaaleja muuttujia.

JavaScript on syntaksiltaan hieman C-kielen kaltainen. C-kielen kaltaisissa kielissä lohko, eli aaltosulkujen sisällä olevat ilmaukset, muodostavat näkyvyysalueen (scope). JavaScriptissä näin ei kuitenkaan ole, vaan jonkin lohkon sisällä määritelty `var`-muuttuja on näkyvillä koko funktion sisällä. Koodiesimerkissä 3.1 nähdään, miten `var`-avainsanalla määritelty muuttuja `a` saa if-lohkon sisällä uuden arvon, joka peittää lohkon ulkopuolella määritellyn muuttujan `a`.

```
function foo() {
  var a = 1;
  if (true) {
    var a = 2;
    console.log(a);
  }
  console.log(a);
}
// tulostaa luvut 2, 2
```

Koodiesimerkki 3.1: `var`-muuttujan näkyvyysalue.

ECMAScript 6 sisältää uuden tavan esitellä muuttujat: `let`- ja `const`-avainsanat. `let` ja `const` määrittelevät muuttujan paikallisesti, eli muuttuja ei ole näkyvissä kuin lohkossa, jossa se on määritelty. `let` määrittelee muuttujan, jonka arvoa on mahdollista muuttaa, `const` taas määrittelee muuttumattoman muuttujan [Int15]. Koodiesimerkissä 3.2 nähdään, että if-lohkon sisällä oleva lokaali muuttuja `a` on näkyvissä ainoastaan lohkon sisällä, eikä peitä lohkon ulkopuolella `var`-avainsanalla määriteltyä muuttujaa `a`.

Samuusvertailu

JavaScriptin huonona ominaisuutena pidetään myös samuusvertailua. Yhtä- ja erisuuruusvertailuja on mahdollista tehdä käyttäen joko abstraktia samuusvertailua (== ja !=) tai tiukkaa samuusvertailua (=== ja !==). Abstrakti samuusvertailu muuttaa operandit ensin saman tyyppisiksi ja tarkistaa vas-

ta sitten, vastaavatko operandien sisällöt toisiaan. Tiukka samuusvertailu puolestaan tarkistaa, ovatko operandit samaa tyyppiä ja vastaavatko operandien sisällöt toisiaan. Jos operandien tyypit eivät ole samat, tiukka vertailu palauttaa aina arvon `false` [Moz17].

```
function foo() {
  var a = 1;
  if (true) {
    let a = 2;
    console.log(a);
  }
  console.log(a);
}
// tulostaa luvut 2, 1
```

Koodiesimerkki 3.2: Paikallisten muuttujien käyttö näkyvyysalueella.

Koodiesimerkissä 3.3 tehdään yhtäsuuruusvertailuja käyttäen abstraktia vertailua, ja koodiesimerkissä 3.4 käyttäen tiukkaa vertailua samoille operandeille. Esimerkeissä havainnollistuu abstraktin ja tiukan vertailun ero. Yleisesti pidetään hyvänä käytänteenä käyttää tiukkaa vertailua abstraktin vertailun sijaan.

```
1 == '1'    // true
1 == 1      // true

[] == ![]   // true
```

Koodiesimerkki 3.3: Abstrakti samuusvertailu.

```
1 === '1'    // false
1 === 1      // true

[] === ![]   // false
```

Koodiesimerkki 3.4: Tiukka samuusvertailu.

Automaattinen puolipisteen sijoitus

JavaScriptissä jotkin lauseet vaativat puolipisteen lauseen päättymisen merkiksi. Tämän vuoksi JavaScriptissä on automaattinen puolipisteen sijoitusominaisuus (automatic semicolon insertion, ASI). Tätä ominaisuutta käytetään tyhjille lauseille (pelkkä `;`), muuttujien määrittelylle, `import`- ja `export`-lausekkeissa sekä moduulin määrittelyssä, sijoituslauseissa sekä `debugger`-, `continue`-, `break`-, `throw`- ja `return`-lauseiden yhteydessä [Moz17].

Tähän ominaisuuteen ei kuitenkaan kannata täysin luottaa, sillä joskus ohjelma saattaa automaattisen puolipisteen sijoituksen takia toimia odottamattomalla tavalla. Koodiesimerkissä 3.5 havainnollistetaan tällaista tilannetta. Esimerkissä ohjelmoija voisi olettaa, että halutaan palauttaa objekti, joka sisältää avain-arvo–parin {`foo: 'bar'`}. Palautettava arvo on kuitenkin automaattisen puolipisteen sijoituksen johdosta määrittelemätön, eli `undefined`. Tällaisilta ongelmilta voidaan välttyä, mikäli tässä tapauksessa palautettavan objektin avaava aaltosulku sijoitetaan samalle riville kuin `return`-lause, kuten koodiesimerkissä 3.6.

```
return
{
      foo: 'bar'
};
// kääntäjä lisää return:in perään puolipisteen
// ja palauttaa undefined
```

Koodiesimerkki 3.5: Automaattisen puolipisteen sijoituksen seuraukset.

```
return {
      foo: 'bar'
};
// palauttaa objektin
```

Koodiesimerkki 3.6: Automaattisen puolipisteen sijoituksen seuraukset.

3.4 Työkalut JavaScriptille

JavaScriptille on kehitetty alkuperäistä C-kielen lintiä vastaavia staattisen analyysin tarkistustyökaluja, kuten ESLint [ESL17], JSLint [JSL17] ja JSHint [JSH17], joilla voidaan tarkistaa, että koodista ei löydy syntaksivirheitä ja että koodi täyttää tietyt ohjelmointisäännöt eikä näin ollen sisällä ei-toivottuja huonoja käytänteitä. Näitä työkaluja voidaan käyttää komentoriviltä, tai joissakin tapauksissa myös asentamalla tekstieditoriin lisäosa, jolloin lintin löytämät virheet näytetään ohjelmoijalle suoraan editorissa. Tutkielmassa keskitytään ESLintiin, jota käytetään myös kokeellisessa osiossa.

3.4.1 ESLint

ESLint on JSLintiä ja JSHintiä uudempi työkalu. ESLint on hyvin konfiguroitavissa: käytettävät tarkistussäännöt voidaan konfiguroida täysin sovelluskohtaisesti, tai vaihtoehtoisesti on mahdollista käyttää plugin-tyyppisesti jotakin valmista konfiguraatiota joka sisältää joukon sääntöjä [ESL17]. Valmiin joukon sääntöjä on myös mahdollista muokata sopimaan paremmin tiettyyn sovellukseen.

ESLintin konfigurointi tapahtuu .eslintrc-tiedoston avulla. Koodiesimerkki 3.7 sisältää esimerkin .eslintrc-tiedostosta. Tiedostomuotoja on useita: .js-, .json-, .yaml-, .yml- sekä pelkkä rc-päätteinen. rc-päätteiseen tiedostoon voi määritellä konfiguraation joko JSON- tai YAML-muodossa. Konfiguraatio voidaan myös määritellä suoraan package.json-tiedostoon. Konfiguraatioon voidaan määritellään mitä jäsennintä (parser) käytetään, jos halutaan käyttää jotakin muuta jäsennintä kuin oletuksena olevaa Espreetä. Lisäksi "env"-avaimen alle voidaan määritellä ympäristöt, joiden globaaleja muuttujia halutaan käyttää. Käytettävät säännöt annetaan "rules"-avaimen alle. Jokaiselle säännölle annetaan avaimena säännön nimi, ja arvona taulukko, jossa ensimmäisenä arvona määritellään raportoidaanko sääntörikkomuksesta virhe vai varoitus. Toisena arvona on sääntöön liittyviä mahdollisia

lisävalintoja. ESLintiin voidaan lisäksi määritellä "globals"-avaimen avulla globaaleja muuttujia, joita tietoisesti halutaan käyttää. Tämän avulla ESLint ei raportoi näiden muuttujien käyttämisestä ongelmia.

```
{
  "parser": "babel-eslint",
  "env": {
    "node": true,
    "mocha": true
  },
  "rules": {
    "semi": ["error", "always"]
  }
}
```

Koodiesimerkki 3.7: Esimerkki .eslintrc-tiedostosta

Valmiin sääntöjoukon käyttäminen onnistuu lisäämällä .eslintrc-tiedostoon rivi `"extends": "konfiguraation-nimi"`. Mikäli jokin valmis sääntöjoukko ei ole täysin mieleinen, sen sääntöjä on myös mahdollista ylikirjoittaa .eslintrc-tiedostossa määrittelemällä "rules"-avaimelle uusia sääntöjä tai muokkaamalla vanhoja. Kaikki käytettävät säännöt on mahdollista määritellä myös erikseen .eslintrc-tiedostossa, mikäli mitään valmista konfiguraatiota ei haluta käyttää.

ESLint sisältää suuren joukon sääntöjä, jotka on kuvattu työkalun dokumentaatio-sivuilla verkossa[3]. Säännöt on jaoteltu seitsemään kategoriaan: mahdollisiin virheisiin, hyviin käytänteisiin, muuttujiin, tyyliin, ECMAScript 6:een, "strict"-tilaan sekä Node.js:ään ja CommonJS:ään liittyviin ongelmiin. Mahdolliset virheet viittaavat syntaksi- tai logiikkavirheisiin, kun taas parhaisiin käytänteisiin liittyvät säännöt ohjaavat tekemään asiat paremmalla tavalla, jotta ongelmilta vältyttäisiin. Muuttujiin liittyvät säännöt auttavat muuttujien määrittelyssä. Säännöt, jotka liittyvät tyyliongelmiin

[3] http://eslint.org/docs/rules/

ovat tyyliin liittyviä suosituksia, joten tämän kategorian sääntöjä voidaan pitää hyvin subjektiivisina. ECMAScript 6:een liittyvät säännöt ovat kyseiseen JavaScript-spesifikaatioon liittyviä sääntöjä, jotka siis ohjeistavat käyttämään spesifikaation mukaisia JavaScriptin ominaisuuksia. "strict"-tila sisältää vain yhden säännön, joka liittyy JavaScriptin "strict"-direktiivin käyttöön. Node.js:ään ja CommonJS:ään kuuluvat säännöt liittyvät sananmukaisesti ohjelmakoodiin, jota ajetaan Node.js:llä tai CommonJS:llä.

Sääntörike voidaan määrittää synnyttämään joko virheen tai varoituksen. Varoituksia pidetään virheitä lievempinä rikkomuksina, ja ESLintin suorittaminen on mahdollista tehdä myös siten, että se ei raportoi ollenkaan varoituksia. Lisäksi jos ESLint ei havaitse yhtään virhettä, vaan ainoastaan varoituksia, niin komentorivillä tehty ohjelmakoodin tarkistus päättyy onnistuneeseen lopetukseen (exit with a success exit status).

Koodin tarkistus voidaan suorittaa komentoriviltä analysoitavan kohteen juurikansiosta komennolla "`eslint --ext .js src/`". Kyseinen esimerkkikomento tekee tarkistukset `src`-kansion alta, ottaen huomioon vain .js-päätteiset tiedostot. Se tekee tarkistukset tiedostokohtaisesti, eli eri tiedostojen välisiä riippuvuuksia ESLint ei analysoi. ESLint raportoi havaitut ongelmat komentoriville. Kuvassa 1 on esimerkki siitä, miten ESLint raportoi havaitut ongelmat. Ongelmat raportoidaan tiedostoittain, ja jokainen sääntörikkomus on yksi rivi ja samaa säännön rikkomuksia voi olla usealla eri rivillä. Rivin alussa kerrotaan rikkeen sijainti ohjelmakoodissa. Tämän jälkeen ilmoitetaan onko kyseessä virhe (error) vai varoitus (warning), jota seuraa kuvaus rikkeestä. Rivin lopussa on säännön nimi.

```
/home/sonja/project/backend/src/form.js
   1:1   error    Expected newline after "use strict" directive  lines-around-directive
   1:1   error    'use strict' is unnecessary inside of modules  strict
   2:1   error    Unexpected var, use let or const instead       no-var
   3:1   error    Unexpected var, use let or const instead       no-var
   6:35  warning  Unexpected unnamed function                    func-names
   6:35  error    Unexpected function expression                 prefer-arrow-callback
   6:43  error    Missing space before function parentheses      space-before-function-paren
```

Kuva 1: Esimerkki ESLintin tavasta raportoida ongelmat komentorivillä.

ESLint on myös mahdollista saada näkyville joihinkin tekstieditoreihin, kuten Sublime Textiin lataamalla editoriin lisäosa. Lisäosan avulla ohjelmoija näkee suoraan editorissa tiedostojen koodiriveillä, missä ESLint on havainnut ongelmia, ilman että koodin tarkistusta tarvitsee suorittaa komentoriviltä.

3.4.2 Muut lint-työkalut

JSLint on valmiiksi konfiguroitu lint-työkalu, ja se sisältää valmiit säännöt, joiden mukaan koodia analysoidaan [JSL17]. JSHint on JSLintin pohjalta tehty konfiguroitavampi versio, jossa käyttäjä voi valita mitä sääntöjä koodin tarkistuksessa käytetään [JSH17].

Eräs tutkimusryhmä kehitti JavaScriptille työkalun nimeltä DLint [GPSS15]. DLint on dynaamisen analyysin työkalu, toisin kuin aiemmat JavaScriptin lint-työkalut. DLintiä verrattiin tämän hetken parhaimpiin staattisen analyysin lint-työkaluihin, kuten ESLintiin, JSHintiin ja JSLintiin, ja havaittiin, että se löysi sellaisia virheitä ja ongelmia, joita muut JavaScriptin lint-työkalut eivät löytäneet. Siten hyödyntäen sekä staattista analyysia että dynaamista analyysia, voidaan paremmin löytää mahdolliset ongelmat [GPSS15]. Kuten ESLint [ESL17], JSLint [JSL17] ja JSHint [JSH17], myös DLint on vapaasti saatavilla verkossa[4]. Se ei kuitenkaan näyttäisi saavuttaneen kovin suurta suosiota Githubissa projektista luotujen uusien kopioiden (fork) ja arvostuksen (star) lukumäärien perusteella.

4 Tutkimusasetelma

Tutkimuksen tavoitteena on mitata sitä, miten ESLint-työkalun käyttöönotto projektissa vaikuttaa sovelluksen ylläpidettävyyteen. Tutkimus tehdään yksittäiselle projektille, jossa otettiin ESLint käyttöön. Käyttöönoton pohjalta pohditaan myös, mitä hyötyjä ESLintistä on, ja miten se saadaan helposti osaksi kehitysprosessia.

[4] https://github.com/Berkeley-Correctness-Group/DLint

Ylläpidettävyyttä mitataan ennen kuin ESLintin havaitsemia ongelmia on korjattu, ja sen jälkeen kun korjaukset on tehty. Ylläpidettävyyden analysointi tapahtuu SonarQuben avulla. Lisäksi muuttunutta ohjelmakoodia käydään läpi kohdeprojektin kehittäjän kanssa arvioiden, onko ESLintin raportoimien ongelmien korjaaminen vaikuttanut koodin ylläpidettävyyteen kehittäjän näkökulmasta. Yksikkö- ja hyväksymätestit suoritetaan ennen ja jälkeen korjausten, jotta varmistutaan siitä, että korjatut ongelmat eivät muuta sovelluksen toimintaa.

4.1 Kohdeprojekti

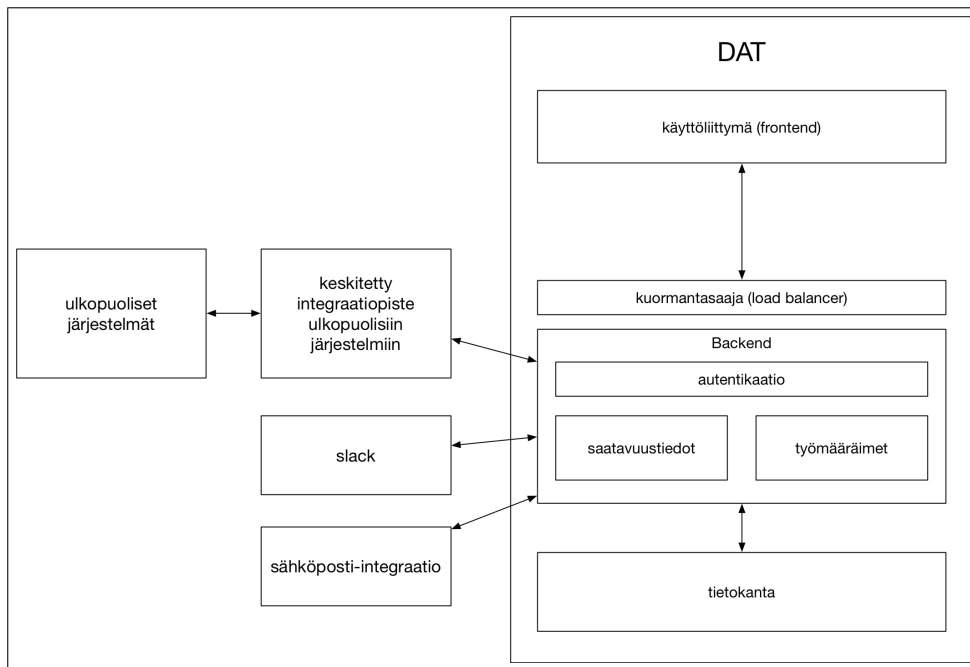
Tutkimuksessa käytetään syksyn 2016 ja kevään 2017 välisenä aikana Kanban-menetelmällä toteutettua projektia. Projektissa kehitettiin työnjärjestelytyökalu, jonka avulla yritykset voivat varata työntekijöitä omiin töihinsä. Projekti julkaistiin keväällä 2017, ja tämän jälkeen sen sivustolle on tehty keskimäärin 3000 vierailua viikossa, joista uniikkeja käyttäjiä on noin 600. Kohdeprojekti sisältää 3429 riviä lähdekoodia. Yhteenveto projektin taustatiedoista on taulukossa 4.1.

Lähdekoodin rivien määrä (LOC/SLOC)	3429 SLOC
Backendin lähdekoodin rivien määrä (LOC/SLOC)	1813 SLOC
Frontendin lähdekoodin rivien määrä (LOC/SLOC)	1616 SLOC
Sovelluksen käyttäjien määrä per viikko	n. 3000 käyttäjää / viikko
Sovelluksen uniikkien käyttäjien määrä per viikko	n. 600 uniikkia käyttäjää / viikko
Julkaisun jälkeisten muutosten määrä	30 koodin kommitointia (commit)

Taulukko 4.1: Yhteenveto kohdeprojektin taustoista.

Kohdeprojektin arkkitehtuuri on esitetty kuvassa 2. Sovelluksen nimi on DAT. Sovellus käsittää käyttöliittymäpuolen (frontend), palvelinpuolen (bac-

kend) ja tietokannan. Lisäksi siinä on integraatioita ulkoisiin järjestelmiin. Sovelluksen frontend on toteutettu Reactilla [Fac17], joka on Facebookin kehittämä avoimen lähdekoodin JavaScript-kirjasto käyttöliittymien toteutusta varten. Backend on toteutettu Koa.js:llä [Koa17]. Koa.js on ohjelmistokehys Node.js:llä toteutetuille web-sovelluksille. Node.js [Fou17b] on avoimen lähdekoodin JavaScript-sovellusalusta, joka mahdollistaa JavaScriptin käytön myös palvelinpuolella.



Kuva 2: Kohdeprojektin arkkitehtuuri.

Tutkimuksessa keskitytään sovelluksen backendiin, josta ESLintin antamat virheet korjataan ja jonka ylläpidettävyyttä analysoidaan.

4.2 ESLintin konfigurointi

Projektin backendissä käytetään Airbnb:n toteuttamaa konfiguraatiota [Air17] ESLintille. Konfiguraation saa käyttöön asentamalla projektiin paketin eslint-config-airbnb-base JavaScriptille tarkoitetun paketinhallintajärjestel-

mä npm:n avulla. Myös itse ESLint asennetaan paketinhallintajärjestelmän avulla. ESLintistä käytettiin versiota 3.19.0 ja eslint-config-airbnb-base:sta versiota 11.2.0.

Kun eslint-config-airbnb-base on asennettu, määritellään projektin (tässä tapauksessa kohdeprojektin backendin) juurikansiossa olevaan .eslintrc-konfiguraatiotiedostoon käytettävä konfiguraatio. Tiedostossa määritellään käytettäväksi Airbnb:n konfiguraatiota lisäämällä sinne rivi `"extends": "airbnb-base"`. Tämän lisäksi muutamaa Airbnb:n konfiguraatiossa olevaa sääntöä muokattiin sopimaan paremmin kyseiselle projektille. Airbnb:n konfiguraatiossa syklomaattisen kompleksisuuden sääntö ei ollut valmiiksi käytössä, joten se lisättiin konfiguraatiotiedostoon. Kohdeprojektin .eslintrc-tiedosto löytyy kokonaisuudessaan liitteestä A.

4.3 Ylläpidettävyyden mittaaminen

Sovelluksen ylläpidettävyyttä mitataan SonarQubella [Son17]. Analyysiin sisällytetään ohjelmakoodia sisältävät tiedostot, jotka sijaitsevat projektin backendin src/-kansiossa.

Tässä tutkielmassa sovellukseen tehdään muutoksia ESLintiin konfiguroitujen sääntöjen perusteella, joten SonarQube konfiguroitiin analysoimaan sovellusta näitä samoja sääntöjä käyttäen. ESLintin säännöt saa käyttöön SonarQubessa SonarQube-ESLint-pluginin[5] avulla. Pluginin avulla projektin .eslintrc-tiedostoon määritellyt säännöt saadaan vietyä automaattisesti SonarQubeen. Käytetty SonarQuben konfiguraatiotiedosto löytyy liitteestä B.

SonarQubessa mittareiksi valittiin kompleksisuuteen, toisteisuuteen, sovelluksen kokoon ja testeihin liittyvät mittarit. Esimerkiksi koodihajuihin ja tekniseen velkaa liittyvät mittarit jätettiin tästä pois, koska näihin vaikuttavat suoraan se, että ESLintin havaitsemat ongelmat korjataan, jolloin sekä

[5] `https://github.com/sleroy/SonarEsLintPlugin`

koodihajut että tekninen velka häviävät. SonarQubesta käytettiin versiota 5.6.6 ja SonarJS:stä versiota 3.2.0.

SonarQubella tehtävän laatuanalyysin lisäksi sovelluksen tehtiin ohjelmakoodiin katselmointi, jossa arvioitiin muutoksia ja ylläpidettävyyttä sovelluskehittäjän näkökulmasta.

5 Tulokset

Tässä luvussa raportoidaan säännöt, joihin liittyen ESLint havaitsi ongelmia kun ESLint otettiin käyttöön eikä ongelmia oltu vielä korjattu. Lisäksi esitellään SonarQuben analyysin tulokset sekä kohdeprojektin kehittäjän näkemys ESLintin käyttöönotosta.

5.1 ESLintin havaitsemat ongelmat

ESLint havaitsi backendin ohjelmakoodissa yhteensä 979 ongelmaa, joista 874 kategorisoitiin virheiksi ja 105 varoituksiksi. Virheet rikkoivat 36:a eri sääntöä, ja varoitukset kahta eri sääntöä. Säännöt, joihin virheet ja ongelmat liittyivät, kuuluivat viiteen kategoriaan: mahdollisiin virheisiin, hyviin käytänteisiin, "strict"-tilan käyttöön, tyyliin sekä ECMAScript 6:een.

Viisi eniten virheitä tuottanutta ESLintin sääntöä, sekä virheiden lukumäärä per sääntö on listattuna taulukossa 5.1. Täydellinen listaus kaikista virheitä aiheuttaneista säännöistä löytyy liitteestä C.

Viisi eniten virheitä tuottanutta sääntöä ovat "no-var" (204 virhettä), "comma-dangle" (111 virhettä), "generator-star-spacing" (107 virhettä), "object-curly-spacing" (104 virhettä) ja "prefer-const" (57 virhettä). "no-var", "generator-star-spacing" ja "prefer-const" kuuluvat ECMAScript 6 –kategoriaan, "object-curly-spacing" tyyli-kategoriaan ja "comma-dangle" mahdollisten virheiden kategoriaan.

Sääntö	**Kategoria**	**Kuvaus**	**Virheet lkm**
no-var	ECMAScript 6	Vaatii käyttämään muuttujien määrittelyssä avainsanan `var` sijaan `const` tai `let`.	204
comma-dangle	Mahdolliset virheet	Pakottaa yhdenmukaiseen pilkkujen käyttöön objektiliteraaleissa. Käytetty konfiguraatio vaatii pilkun objektin viimeisen elementin tai ominaisuuden jälkeen ainoastaan, mikäli sulkeva haka- tai aaltosulku on eri rivillä kuin objektin viimeinen elementti tai ominaisuus. Sääntö kieltää objektin viimeisen elementin tai ominaisuuden jälkeisen pilkun, mikäli elementti ja sulkeva haka- tai aaltosulku ovat samalla rivillä.	111
generator-star-spacing	ECMAScript 6	Määrittelee yhdenmukaisen tavan käyttää välilyöntejä *-merkin ympärillä generaattorifunktioissa. Käytetty konfiguraatio märittelee, että välilyönti tulee olla *-merkin jälkeen mutta ei ennen.	107
object-curly-spacing	Tyyli	Pakottaa yhdenmukaiseen välilyöntien käyttöön aaltosulkujen sisällä. Käytetyssä konfiguraatiossa pakotetaan aina käyttämään välilyöntejä aaltosulkujen sisällä, lukuunottamatta tyhjää lohkoa "{}".	104
prefer-const	ECMAScript 6	Pakottaa määrittelemään muuttujat `let`-avainsanan sijaan `const`-avainsanalla, mikäli muuttujalle ei myöhemmin aseteta uutta arvoa.	57

Taulukko 5.1: Viisi eniten virheitä aiheuttanutta ESLint sääntöä.

Säännöt, joista ESLint raportoi varoituksen sekä varoitusten lukumäärä on listattuna taulukossa 5.2. ESLint antoi varoituksia kahden säännön rikkomisesta. Näistä tyyli-kategoriaan kuuluvaa "func-names"–sääntöä, joka varoittaa nimeämättömistä funktioista, rikottiin 97 kertaa ja mahdollisten virheiden kategoriaan kuuluvaa "no-console"–sääntöä kahdeksan kertaa.

Sääntö	**Kategoria**	**Kuvaus**	**Varoitukset lkm**
func-names	Tyyli	Pakottaa tai kieltää nimettyjen funktioiden käytön. Nimettyjen funktioiden käyttö helpottaa vikojen selvittämistä, kun kutsupinossa (stack trace) näkyy anonyymin funktion sijaan nimetty funktio. Käytetty konfiguraatio varoittaa aina nimeämättömistä funktioista.	97
no-console	Mahdolliset virheet	Kieltää console-objektin metodeihin tehtävät kutsut	8

Taulukko 5.2: ESLintin havaitsemat varoitukset.

Kaikki ESLintin raportoimat virheet ja varoitukset korjattiin. Korjausten tekemiseen kului noin kaksi henkilötyöpäivää.

5.2 Ylläpidettävyysmittausten tulokset

Kohdeprojektin backend analysoitiin kun ESLint oltiin otettu käyttöön, mutta yhtään korjausta ei oltu vielä tehty, ja uudestaan sen jälkeen kun kaikki ESLintin havaitsemat ongelmat oli korjattu. SonarQubella tehtyjen laatuanalyysien tulokset löytyvät taulukosta 5.3. Taulukossa näkyy analyysin tulokset ennen ja jälkeen ESLintin havaitsemien ongelmien korjaamista.

Kuten taulukosta 5.3 nähdään, yleisesti ottaen ohjelmakoodin kompleksisuus, toisteisuus sekä koodirivien määrä kasvoi. Ohjelmakoodin kognitiivisen

kompleksisuuden arvo (taulukon kompleksisuusarvo) kasvoi kuudella: luvusta 255 lukuun 261. Myös kompleksisuus per tiedosto kasvoi 9,8:sta 10,0:een, mutta kompleksisuus per funktio pieneni 2,1:stä 2,0:een.

Metriikka	**Ennen**	**Jälkeen**
Kompleksisuusarvo	255	261
Keskimääräinen kompleksisuusarvo per tiedosto	9,8	10,0
Keskimääräinen kompleksisuusarvo per funktio	2,1	2,0
Toisteisten rivien osuus ohjelmakoodissa	3,1 %	5,3 %
Toisteisuutta sisältävien tiedostojen määrä	2	4
Toisteisuutta sisältävien lohkojen määrä	5	7
Toisteisuutta sisältävien rivien määrä	83	146
Koodirivien määrä	1813	1877
Funktioiden määrä	114	121
Testien kattavuus	55,5 %	56,0 %
Rivikattavuus	58,9 %	59,4 %

Taulukko 5.3: SonarQuben tulokset ennen ja jälkeen ESLintin mukaisia korjauksia.

Toisteisten rivien osuus ohjelmakoodissa kasvoi korjausten myötä noin 71 % eli 3,1 prosentista 5,3 prosenttiin. Toisteisuutta sisältäviä lohkoja havaittiin 40 % enemmän (ennen korjauksia 5, korjausten jälkeen 7), toisteisuutta sisältäviä rivejä 75,9 % enemmän (ennen korjauksia 83, korjausten jälkeen 146) ja toisteisuutta sisältäviä tiedostoja 100 % enemmän (ennen korjauksia 2, korjausten jälkeen 4).

Koodirivien määrä puolestaan kasvoi noin 3,5 % (ennen korjauksia 1813, korjausten jälkeen 1877). Myös funktioiden määrä kasvoi noin 6,1 %, 114:stä 121:een. Lisäksi sovelluksen testikattavuus parani hieman: 55,5 prosentista 56,0 prosenttiin.

5.3 Kehittäjän näkemys ESLintin käyttöönotosta

Kun ESLintin raportoimat virheet oli korjattu, tehtiin ohjelmakoodille katselmointi, johon osallistui yksi projektissa mukana olleista kehittäjistä. Tässä aliluvussa käydään läpi katselmoinnissa esille tulleita asioita.

Yhdenmukaista ja samannäköistä koodia pidettiin ylläpidettävyyttä parantavana tekijänä. Lisäksi kun koodi noudattaa alan standardeja ja yleisesti hyvinä pidettäviä käytäntöjä (best practices), uusien kehittäjien on helpompi ymmärtää koodia ja tulla mukaan kehitystyöhön. Yhdenmukaisuuden noudattamisen hyötynä kehittäjä näki myös sen, että muutosten tekeminen on helpompaa, kun eri tiedostojen ja moduulien välillä koodin kirjoitusasu ei muutu. Koska JavaScript on vapaasti muotoiltava kieli, etenkin laajemmissa projekteissa kehittäjä piti tyylisääntöjen noudattamista välttämättömänä.

Kehittäjä piti ESLintiä hyvänä työkaluna esittelemään JavaScriptin uusia standardeja ja ohjaamaan niiden käyttöä. ES6-standardi sisältää paljon uusia ja tehokkaampia ilmaisutapoja asioille, jotka näyttivät aikaisemmin vaikeaselkoisemmilta. Esimerkiksi koodinpätkä koodiesimerkissä 5.1 muotoutui ESLintin virheraportointien avulla hieman luettavampaan muotoon (koodiesimerkki 5.2), kun ilmaisutapana käytettiin ES6:n myötä tullutta "arrow-callback"-tyyliä.

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});
```

Koodiesimerkki 5.1: Koodipätkä kohdeprojektista ennen korjauksia.

```
passport.serializeUser((user, done) => done(null, user.id));
```

Koodiesimerkki 5.2: Koodipätkä kohdeprojektista korjausten jälkeen.

ESLint raportoi eniten virheitä `var`-avainsanan käytöstä. Hyvänä puolena `let`- ja `const`-avainsanojen käytössä `vari`:in sijaan pidettiin sitä, et-

tä ne tuovat JavaScriptiin funktionaalisen ohjelmoinnin hyviä periaatteita muuttujien muuttumattomuuden muodossa. Siten `var`:in korvaamista `let`:lla tai `const`:lla pidettiin hyvänä asiana, sillä näiden käyttö turvaavat hyvän suorituskyvyn sekä riippuvuuksien ja vastuiden selkeämmän määrittelyn moduuleissa ja funktioissa.

Lisäksi katselmoinnissa tuli esille, että ohjelmistokehityksessä organisaatiolla tulisi olla selkeä pohja ESLintin käyttöönottoon ja hyödyntämiseen osana jatkuvaa laadunvarmistusta. Pohjaa voitaisiin käyttää esimerkiksi osana tuoreena pidettävää "template"-projektia, josta luodaan uusia projekteja. Tällöin ESLint tulisi automaattisesti käyttöön alusta alkaen kaikkiin projekteihin valmiiksi konfiguroituna.

6 Keskustelu

Tässä luvussa analysoidaan saatuja tuloksia. Ensimmäisessä aliluvussa arvioidaan ESlintin havaitsemia ongelmia ja niiden korjauksista seuraavia muutoksia. Tämän jälkeen keskustellaan ESLintin käyttöönotosta ohjelmistoprojektissa. Lopuksi pohditaan mahdollisia rajoitteita ja jatkotutkimuskohteita sekä vastataan tutkimuskysymyksiin.

6.1 ESLintin tulosten arviointi

Aiemmin esitellyt ongelmat JavaScriptin ominaisuuksissa, eli globaalit muuttujat ja näkyvyysalue, samuusvertailu ja automaattinen puolipisteen sijoitus tulivat esille myös ESLintin havaitsemissa ongelmissa. JavaScriptin globaaleihin muuttujiin ja näkyvyysalueeseen liittyen ESLint havaitsi ongelmia `var`-avainsanan käytön yhteydessä (204 virhettä, `no-var`-sääntö) sekä muuttujien määrittelyssä (41 virhettä, `vars-on-top`-sääntö). Automaattisen puolipisteen sijoitukseen liittyen havaittiin 13 virhettä (`semi`-sääntö) ja samuusvertailu-ongelmaan liittyen yksi virhe (`eqeqeq`-sääntö).

ESLint antoi useita varoituksia nimeämättömien funktioiden käytöstä. Nimeämättömien funktioiden muuttaminen nimetyiksi voidaan katsoa parantavan ylläpidettävyyttä, sillä jos sovellukseen tulee vikoja, niiden selvittäminen helpottuu, kun kutsupinosta näkee suoraan mitä nimettyä funktiota on kutsuttu.

ESLintiin oli määritelty funktion syklomaattisen kompleksisuuden rajaksi kahdeksan. Kyseisen säännön rikkeitä havaittiin kaksi. Monimutkaiset funktiot jaettiin useammaksi funktioksi, jolloin kompleksisuus pieneni sallitulle tasolle korjausten myötä. Tämän perusteella voidaan sanoa, että funktioiden kompleksisuus pieneni, mitä voidaan pitää ylläpidettävyyden kohentumisena. On kuitenkin huomattava, että vaikka syklomaattinen kompleksisuus per funktio pieneni, SonarQuben laskema koko sovelluksen kognitiivinen kompleksisuus puolestaan kasvoi kaikkien ESLint-korjausten myötä arvosta 255 arvoon 261. Kognitiivinen kompleksisuusarvo per funktio kuitenkin pieneni hieman, arvosta 2,1 arvoon 2,0. ESLintin avulla ei siis voida vaikuttaa kokonaiskompleksisuuteen, mutta yksittäisten funktioiden kompleksisuuteen voidaan.

Vaikka SonarQuben analysoima toisteisuuden prosentuaalinen muutos oli suuri, niin korjausten jälkeinen toisteisuuden määrä pysyi kuitenkin suhteellisen pienenä. Toisteisuuden lisääntyminen saattaa kuitenkin huonontaa sovelluksen ylläpidettävyyttä ja kehittäjien olisi hyvä ottaa huomioon lisääntynyt toisteisuus ja mahdollisesti refaktoroida tämä pois. Lisäksi on huomioitava, että ESLint analysoi ohjelmakoodia tiedosto kerrallaan, joten se ei voi havaita eri puolilla sovellusta olevaa toisteisuutta. Toisteisuuden kitkemiseksi on siis käytettävä jotakin muuta työkalua. ESLintin käytöstä seuraavan koodin yhdenmukaisuuden avulla esille voi siis tulla muita piileviä ongelmia, kuten toisteisuus, jota SonarQube ei aiemmin havainnut.

Koodirivien määrä kasvoi hieman kahdesta syystä. Ensinnäkin kahden funktion kompleksisuus oli ESLintin sääntöjen mukaan liian korkea, ja nämä

pilkottiin osiin. Tämän johdosta koodirivejä oli korjausten jälkeen enemmän. Lisäksi sovelluksen koodissa oli sellaisia rivejä, joiden pituus ylitti sallitun 200 merkin rajan. Tällaiset rivit sisälsivät useimmiten taulukkomuotoista dataa. Koodirivin pituus saatiin sallittuihin rajoihin muuttamalla taulukon esitysmuotoa siten, että jokainen alkio on omalla rivillään. Koodiesimerkissä 6.1 on esimerkkitapaus siitä, miten tämänkaltaiset koodirivin pituusylitykset korjattiin.

```
// Toteutus ennen korjauksia (kaikki samalla rivillä):
var attributes = form(this.request.body, ['firstname', '
    lastname', 'phone', 'employeeId', 'address', 'city', '
    postalcode', 'primarycharacteristic', 'serviceterritory', '
    description'], { organizationId: user.organizationId });


// Toteutus korjausten jälkeen:
const attributes = form(this.request.body,
    [
      'firstname',
      'lastname',
      'phone',
      'employeeId',
      'address',
      'city',
      'postalcode',
      'primarycharacteristic',
      'serviceterritory',
      'description',
    ],
    { organizationId: user.organizationId }
  );
```

Koodiesimerkki 6.1: Koodirivin pituuden lyhentäminen ja sen vaikutukset koodirivien määrään.

Vaikka lisääntynyttä koodirivien määrää pidetään huonona merkkinä ylläpidettävyyden suhteen, tässä tapauksessa muutos ei välttämättä tätä tarkoita. Kun esimerkiksi pitkän yhdellä rivillä esitetyn taulukon esittää monirivisenä, se helpottaa koodin luettavuutta ja muutoksia tehdessä virheitäkään tuskin tulee tehtyä yhtä helposti, kun koko koodinpätkä on nähtävillä joutumatta vierittää tekstiä editorissa.

Testien kattavuus parani hieman lisääntyneiden koodirivien vuoksi. Koska lisääntyneet koodirivit koskivat jo aiemmin toteutetun ominaisuuden ilmaisua luettavammassa ja siistimmässä muodossa, niin vaikka testit kattoivatkin edelleen saman määrän toimintoja, katettujen toimintojen rivimäärä kasvoi, jolloin myös kattavuus kasvoi.

ESLintissä käytettiin "comma-dangle"-sääntöä, jonka rikkomista havaittiinkin paljon (111 kertaa). Sääntö vaatii, että esimerkiksi taulukoissa, jos esitystapa on yksi alkio per rivi, niin viimeisen alkion perään lisätään pilkku (esimerkiksi koodiesimerkin 6.1 korjausten jälkeinen toteutus: pilkku alkion `'description'` jälkeen). Tällöin muutosten tekeminen – kuten uuden alkion lisääminen taulukkoon – helpottuu: muutos koskee ainoastaan uutta lisättävää riviä, sillä edellisellä rivillä on jo pilkku. Näin ollen esimerkiksi `git diff` -ominaisuutta käytettäessä muuttuneen koodin tutkiminen on helpompaa, sillä yhden koodirivin lisääminen myös näkyy vain yhden rivin muuttumisena. Jos pilkku joudutaan lisäämään edelliselle riville muutoksen yhteydessä, yhden koodirivin lisääminen vaikuttaakin kahteen riviin.

Lint-työkalun käytön voidaan tulosten perusteella katsoa parantavan JavaScript-sovelluksen ohjelmakoodin laatua, ja näin ollen myös ylläpidettävyyttä. Lisäksi yhdenmukainen koodaustyyli auttaa tuomaan näkyville esimerkiksi toisteisuutta, joka muuten voisi jäädä huomaamatta. Jos ohjelmistoprojektissa on käytössä jokin valmis konfiguraatio lintille, kuten Airbnb:n toteuttamat ESLintin suosituskonfiguraatiot, lintin avulla voi olla mahdollista myös varmistaa, että sovelluksessa käytetään yhteneväisesti kaikkialla

samaa JavaScript-versiota. Tämä edesauttaa kehittäjien tietoisuutta uusien JavaScript-versioiden tuomista ominaisuuksista sekä ohjaa käyttämään niitä.

6.2 ESLintin käyttöönotto ohjelmistoprojektissa

ESLintin käyttöönotto on suhteellisen helppoa, etenkin jos käyttää jotakin valmista konfiguraatiota. Tutkielman kokeellinen osio osoitti, että havaittujen ongelmien korjaaminen on suhteellisen nopeaa, mikä myös puoltaa ESLintin käyttöönoton kannattavuutta. Etenkin ESLintin kohdalla voitaisiinkin sanoa, että aiemmissa tutkimuksissa havaittu staattisen analyysin tuoma edullinen tapa lisätä laadunvarmistusta pätee.

Helpoiten ESLintin käyttö saadaan osaksi kehitystyötä käyttämällä tekstieditoreille tehtyjä plugineja. Editori-pluginin avulla kehittäjä näkee reaaliaikaisesti koodia lukiessaan rivit, jotka eivät täytä asetettuja sääntöjä sekä virheen tai varoituksen sisältämän viestin. Lisäksi kehityksessä voisi olla hyvä ottaa käytännöksi suorittaa koodin tarkistukset komentoriviltä esimerkiksi aina ennen kuin koodimuutoksia kommitoidaan ja viedään versionhallintaan. Koodin yhdenmukaisuutta voitaisiin seurata myös esimerkiksi lisäämällä ESLintin tarkistukset osaksi jatkuvaa integraatiota (continuous integration, CI) joko esimerkiksi suorittamalla tarkistukset build-palvelimella tai käyttämällä ESLintin sääntöjä laatuanalyysissa esimerkiksi SonarQubessa.

ESLintin (tai jonkin muun tyylintarkistustyökalun) käytöstä voitaisiin hyötyä vieläkin enemmän, jos työkalu otetaan käyttöön organisaatiotasolla, eli kaikissa JavaScriptillä toteutettavissa projekteissa. Tämä voidaan toteuttaa esimerkiksi siten, että käytäntönä on, että jokaisessa alkavassa JavaScript-projektissa on käytössä ESLint heti alusta lähtien. Kaikissa projekteissa voidaan käyttää samaa pohja-konfiguraatiota, jota on mahdollista muokata vastaamaan aina yksittäisen projektin tarpeita. Näin tyyli kirjoittaa JavaScriptiä on yhtenäinen kaikissa projekteissa, mikä helpottaa kehittäjän siirtymistä toiseen projektiin, ja projektin siirtymistä ylläpitovaiheeseen.

6.3 Rajoitteet ja mahdolliset jatkotutkimuskohteet

Koe tehtiin yhdelle yksittäiselle sovellukselle. Käytössä oli yksi lint-työkalu ja sen tietty konfiguraatio, joten eri työkalujen tai konfiguraation käyttö olisi voinut tuottaa erilaiset tulokset. Jos käytetyt ESLintin säännöt olisivat olleet erilaiset, se olisi havainnut erilaiset ongelmat, joita olisi myös ollut eri määrä.

Lint-työkalun hyödyt suhteessa ylläpidettävyyteen näkyvät paremmin vasta pidemmällä aikavälillä. Yleistettävämpien tulosten saavuttamiseksi tutkimus voitaisiin toteuttaa laajemmalla skaalalla, testaten esimerkiksi kymmenien tai satojen JavaScript-sovellusten ylläpidettävyyttä ennen ja jälkeen lint-työkalun käyttöönoton sekä pidemmällä aikavälillä. Koodin ylläpidettävyyteen saattaa vaikuttaa paljon se, kuinka kokenut ohjelmoija on, ja lint-työkalun käyttöönottaminen ei välttämättä kaikissa tapauksissa vaikuta samalla tavalla ylläpidettävyyteen. Esimerkiksi kokeneempi ohjelmoija todennäköisesti kykenee rutiininomaisesti tuottamaan ylläpidettävämpää koodia kuin vasta ohjelmoinnin aloittanut henkilö, jolloin kokeneen ohjelmoijan toteuttamassa sovelluksessa lintin käyttäminen tuskin parantaa ennestään hyvää ylläpidettävyyttä. Tässä tapauksessa ESLintin käyttämisestä ei todennäköisesti ole haittaakaan, ja sen käyttämisen avulla saattaa karsiutua inhimillisiä virheitä, kuten puolipisteen unohtaminen rivin lopusta.

6.4 Tutkimuskysymyksiin vastaaminen

Tutkielma vastaa tutkimuskysymyksiin seuraavasti:

TK1: *Miten ESLintin käyttäminen vaikuttaa sovelluksen ylläpidettävyyteen?*

Yhdenmukaisen koodaustyylin katsotaan vaikuttavan positiivisesti sovelluksen ylläpidettävyyteen. ESLintillä voidaan lisäksi rajoittaa funktioiden monimutkaisuutta, jolla on myös positiivinen vaikutus ylläpidettävyyteen. Kokonaiskompleksisuutta ESLint ei näytä kuitenkaan parantavan, vaan päin-

vastoin se voi hieman kasvattaa sitä. ESLintin käyttöönoton jälkeen sovelluksen toisteisuus kasvoi, mutta toisaalta tätä voidaan pitää hyvänä asiana: ilman yhdenmukaista koodaustyyliä toisteisuus ei olisi tullut esille, ja tätä kautta ylläpidettävyyttä voi olla helpompi parantaa.

TK2: *Onko ESLintin käyttämisestä hyötyä ohjelmistokehityksessä?*
Tämän tutkimuksen perusteella voidaan sanoa, että ESLintistä on hyötyä kehityksessä. Etenkin useamman kehittäjän tehdessä samaa projektia, ESLintin avulla koodi saadaan pidettyä yhdenmukaisena. Lisäksi ESLintin raportoimat ongelmat ovat suhteellisen nopeita korjata.

TK3: *Miten ESLint saadaan osaksi kehitysprosessia?*
Helpointa ESLintin käyttö kehitysprosessissa on, jos kehittäjät saavat ESLintin näkymään tekstieditoreissaan, jolloin ongelmat näkyvät suoraan niitä koskevilla koodiriveillä. ESLint voitaisiin myös ottaa osaksi jatkuvaa integraatiota. Lisäksi yrityksen laajuinen käytäntö voisi olla, että JavaScript projekteissa käytetään jotakin hyväksihavaittua pohjakonfiguraatiota ESLintille, jota voidaan tarpeen mukaan soveltaa projektikohtaisesti. Tällöin kaikissa projekteissa on samanlainen koodaustyyli, joka helpottaa muun muassa kehittäjien siirtymistä projektien välillä.

7 Yhteenveto

Sovellusten ylläpitäminen on usein kallista. Näin ollen on tärkeää, että sovelluksen laatu on hyvä, eikä ylläpidettävyys pääse heikkenemään. Ylläpidettävyyttä voidaan mitata muun muassa sovelluksen kompleksisuutta tai kokoa kuvaavilla mittareilla. Myös teknisellä velalla ja koodihajuilla voi olla vaikutusta ylläpidettävyyteen. Ohjelmakoodia voidaan analysoida staattisen tai dynaamisen analyysin työkaluilla, joilla voidaan havaita sovelluksesta

muun muassa muistin käyttöön tai turvallisuuteen liittyviä ongelmia. Joillakin työkaluilla on myös mahdollista tehdä ylläpidettävyyteen ja yleisestikin sovelluksen laatuun liittyviä mittauksia.

Etenkin JavaScriptiä käyttävissä sovelluksissa on mahdollista tehdä toteutuksia monilla eri tavoin, jolloin ohjelmakoodia voi olla hankalaa pitää yhdenmukaisena. Jos ohjelmakoodissa ei noudateta mitään yhtenäistä tyyliä, esimerkiksi uuden kehittäjän voi olla vaikea saada selkoa vanhoista toteutuksista. JavaScript sisältää myös joitakin sellaisia ominaisuuksia, joiden katsotaan olevan huonoja, ja joita ohjelmoijan tulee välttää. Niinpä JavaScriptille on kehitetty erilaisia lint-työkaluja, jotka tarkistavat muun muassa ohjelmakoodin yhdenmukaisuutta, sekä havaitsevat mahdollisia virheitä.

Tämän tutkielman tavoitteena oli tutkia, vaikuttaako lint-työkalun käyttöönotto JavaScript-sovelluksen ylläpidettävyyteen. Kokeellisessa osuudessa eräässä ohjelmistokehitysprojektissa otettiin käyttöön ESLint, ja verrattiin vaikuttaako ESLintin havaitsemien ongelmien korjaus sovelluksen ylläpidettävyyteen. ESLint on työkalu, joka erikseen määritellyn säännöstön perusteella tekee ohjelmakoodille staattista analyysia. Tuloksista ilmeni, että ESLintin käyttöönotto vaikutti sovelluksen ylläpidettävyyteen positiivisesti. ESLint on myös helppo ottaa käyttöön, ja sen havaitsemien ongelmien korjaaminen on melko nopeaa. Parhaiten ohjelmistokehitysfirmat voisivat hyötyä ESLintistä käyttämällä kaikissa JavaScript-projekteissa yhteneväistä säännöstöä. ESLint ei toki ratkaise kaikkia ongelmia, mutta sen avulla kehittäjä voidaan ohjata kirjoittamaan JavaScriptiä oikein ja yhdenmukaisesti, ja sen avulla ohjelmakoodista voidaan saada esille muita piileviä ongelmia, kuten toisteisuutta.

Lähteet

[Act17] Actcat, Inc: *SideCI*, 2017. `https://sideci.com/`, vierailtu 2017-11-26 .

[Air17] Airbnb: *Airbnb base config for ESLint*, 2017. `https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb-base`, vierailtu 2017-08-09 .

[AMDM+16] Alves, N.S.R., Mendes, T.S., De Mendonça, M.G., Spinola, R.O., Shull, F. ja Seaman, C.: *Identification and management of technical debt: A systematic mapping study.* Information and Software Technology, 70:100–121, 2016.

[BB01] Boehm, B. ja Basili, V. R.: *Software Defect Reduction Top 10 List.* Computer, 34(1):135–137, tammikuu 2001, ISSN 0018-9162.

[Che17a] Checkmarx, Ltd: *Checkmarx*, 2017. `https://www.checkmarx.com/`, vierailtu 2017-11-26 .

[Che17b] Checkstyle: *Checkstyle*, 2017. `http://checkstyle.sourceforge.net/`, vierailtu 2017-11-26 .

[Cri17] CriticalBlue: *Prism*, 2017. `https://www.prism-services.io/`, vierailtu 2017-11-29 .

[Cro08] Crockford, D.: *JavaScript: The Good Parts.* O'Reilly Media, Inc., 2008.

[Cun92] Cunningham, Ward: *The WyCash Portfolio Management System.* SIGPLAN OOPS Mess., 4(2):29–30, joulukuu 1992, ISSN 1055-6400.

[EA+97] ECMAScript, ECMA, Association, European Computer Manufacturers *et al.*: *Ecmascript language specification*, 1997.

[ESL17] *ESLint*, 2017. `https://www.eslint.org/`, vierailtu 2017-05-19 .

[Fac17] Facebook, Open Source: *React*, 2017. `https://facebook.github.io/react/`, vierailtu 2017-06-21 .

[FB99] Fowler, Martin ja Beck, Kent: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[Fou17a] Foundation, Eclipse: *Eclipse*, 2017. `https://www.eclipse.org/ide/`, vierailtu 2017-11-26 .

[Fou17b] Foundation, Node.js: *Node.js*, 2017. `https://www.nodejs.org/en/`, vierailtu 2017-06-21 .

[Fow06] Fowler, Martin: *CodeSmell*, 2006. `https://martinfowler.com/bliki/CodeSmell.html`, vierailtu 2017-08-03 .

[GPSS15] Gong, Liang, Pradel, Michael, Sridharan, Manu ja Sen, Koushik: *DLint: Dynamically Checking Bad Coding Practices in JavaScript.* Teoksessa *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, sivut 94–105, New York, NY, USA, 2015. ACM, ISBN 978-1-4503-3620-8.

[Gui83] Guimaraes, Tor: *Managing Application Program Maintenance Expenditures.* Commun. ACM, 26(10):739–746, lokakuu 1983, ISSN 0001-0782.

[Hal77] Halstead, Maurice H.: *Elements of software science.* Elsevier computer science library, Operating and programming systems series. Elsevier, New York (NY), 1977.

[HHKWB16] Hafiz, Munawar, Hasan, Samir, King, Zachary ja Wirfs-Brock, Allen: *Growing a language: An empirical study on how (and why) developers use some recently-introduced and/or recently-evolving JavaScript features.* Journal of Systems and Software, 121(Supplement C):191 – 208, 2016, ISSN 0164-1212.

[HJ91] Hastings, Reed ja Joyce, Bob: *Purify: Fast detection of memory leaks and access errors.* Teoksessa *In Proc. of the Winter 1992 USENIX Conference*, sivut 125–138. Citeseer, 1991.

[HK07] Huizinga, Dorota ja Kolawa, Adam: *Automated defect prevention: best practices in software management.* John Wiley & Sons, 2007.

[Int15] International, ECMA: *ECMAScript 2015 Language Specification*, 2015. `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`, vierailtu 2017-09-28 .

[Int17] International, ECMA: *ECMAScript 2017 Language Specification*, 2017. `http://www.ecma-international.org/ecma-262/8.0/index.html`, vierailtu 2017-09-28 .

[ISO10] *Systems and software engineering – Vocabulary.* ISO/IEC/IEEE 24765:2010(E), sivut 1–418, joulukuu 2010.

[ISO11] ISO: *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.* ISO/IEC 25010:2011, 2011.

[ISO15] ISO: *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality.* ISO/IEC 25023:2015, 2015.

[Jet17] JetBrains: *IntelliJ IDEA*, 2017. `https://www.jetbrains.com/idea/`, vierailtu 2017-11-26 .

[Joh77] Johnson, Stephen C: *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[JSH17] *JSHint*, 2017. `https://www.jshint.com`, vierailtu 2017-05-21 .

[JSL17] *JSLint*, 2017. `https://www.jslint.com`, vierailtu 2017-05-21 .

[KMB04] Kaner, Cem, Member, Senior ja Bond, Walter P.: *Software Engineering Metrics: What Do They Measure and How Do We Know?* Teoksessa *In METRICS 2004. IEEE CS*. Citeseer, 2004.

[Koa17] Koa.js: *Koa*, 2017. `http://koajs.com/`, vierailtu 2017-06-21 .

[KPG09] Khomh, F., Penta, M. Di ja Gueheneuc, Y. G.: *An Exploratory Study of the Impact of Code Smells on Software Change-proneness*. Teoksessa *2009 16th Working Conference on Reverse Engineering*, sivut 75–84. IEEE, lokakuu 2009.

[LAL15] Li, Z., Avgeriou, P. ja Liang, P.: *A systematic mapping study on technical debt and its management*. Journal of Systems and Software, 101:193–220, 2015.

[LSS07] Linz, Tilo, Schaefer, Hans ja Spillner, Andreas: *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook, 3. painos, 2007, ISBN 1933952083.

[Mai17] Maier, Felix: *Iroh*, 2017. `https://maierfelix.github.io/Iroh/`, vierailtu 2017-11-29 .

[Mar08] Martin, Robert C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1. painos, 2008, ISBN 0132350882, 9780132350884.

[McC76] McCabe, T. J.: *A Complexity Measure*. IEEE Transactions on Software Engineering, SE-2(4):308–320, joulukuu 1976, ISSN 0098-5589.

[Mic17a] Microsoft: *StyleCop*, 2017. `https://github.com/StyleCop`, vierailtu 2017-11-26 .

[Mic17b] Microsoft: *Visual Studio*, 2017. `https://www.visualstudio.com/`, vierailtu 2017-11-26 .

[Mis05] Misra, Subhas Chandra: *Modeling Design/Coding Factors That Drive Maintainability of Software Systems*. Software Quality Journal, 13(3):297–320, 2005, ISSN 1573-1367.

[Moz17] Mozilla: *MDN Web Docs - JavaScript reference*, 2017. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference`, vierailtu 2017-10-18 .

[Mye77] Myers, Glenford J.: *An Extension to the Cyclomatic Measure of Program Complexity*. SIGPLAN Not., 12(10):61–64, lokakuu 1977, ISSN 0362-1340.

[NHN16] Nguyen, Man D., Huynh, Thang Q. ja Nguyen, T. Hung: *Improve the Performance of Mobile Applications Based on Code Optimization Techniques Using PMD and Android Lint*, sivut 343–356. Springer International Publishing, Cham, 2016, ISBN 978-3-319-49046-5.

[RMT09] Riaz, M., Mendes, E. ja Tempero, E.: *A systematic review of software maintainability prediction and metrics*. Teoksessa

2009 3rd International Symposium on Empirical Software Engineering and Measurement, sivut 367–377. IEEE Computer Society, lokakuu 2009.

[SAM12] Sjøberg, Dag I.K., Anda, Bente ja Mockus, Audris: *Questioning Software Maintenance Metrics: A Comparative Case Study.* Teoksessa *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, sivut 107–110, New York, NY, USA, 2012. ACM, ISBN 978-1-4503-1056-7.

[SO16] Stack Overflow, Stack Exchange Inc.: *Developer Survey Results 2016*, 2016. `https://www.insights.stackoverflow.com/survey/2016`, vierailtu 2017-05-19 .

[Som10] Sommerville, Ian: *Software Engineering.* Addison-Wesley Publishing Company, USA, 9. painos, 2010, ISBN 0137035152, 9780137035151.

[Son17] *SonarQube*, 2017. `https://www.sonarqube.org/`, vierailtu 2017-05-19 .

[SW03] Shao, Jingqiu ja Wang, Yingxu: *A new measure of software complexity based on cognitive weights.* Canadian Journal of Electrical and Computer Engineering, 28(2):69–74, huhtikuu 2003, ISSN 0840-8688.

[SYA+13] Sjøberg, D. I. K., Yamashita, A., Anda, B. C. D., Mockus, A. ja Dybå, T.: *Quantifying the Effect of Code Smells on Maintenance Effort.* IEEE Transactions on Software Engineering, 39(8):1144–1156, elokuu 2013, ISSN 0098-5589.

[Syn17] Synopsys, Inc: *Protecode*, 2017. `https://www.synopsys.com/software-integrity/security-testing/`

`software-composition-analysis.html`, vierailtu 2017-11-26 .

[TAV13] Tom, E., Aurum, A. ja Vidgen, R.: *An exploration of technical debt.* Journal of Systems and Software, 86(6):1498–1516, 2013.

[WH88] Wake, S. ja Henry, S.: *A model based on software quality factors which predicts maintainability.* Teoksessa *Proceedings. Conference on Software Maintenance, 1988.*, sivut 382–387. IEEE, lokakuu 1988.

[Wik17] Wikipedia: *ECMAScript*, 2017. `https://en.wikipedia.org/wiki/ECMAScript`, vierailtu 2017-09-28 .

[WOA97] Welker, K.D., Oman, P.W. ja Atkinson, G.G.: *Development and application of an automated source code maintainability index.* Journal of Software Maintenance and Evolution, 9(3):127–159, 1997.

[YHMS15] Yli-Huumo, J., Maglyas, A. ja Smolander, K.: *How do software development teams manage technical debt? - An empirical study.* Journal of Systems and Software, 2015.

[YT89] Young, M. ja Taylor, R. N.: *Rethinking the Taxonomy of Fault Detection Techniques.* Teoksessa *11th International Conference on Software Engineering*, sivut 53–62. ACM, toukokuu 1989.

[ZHB11] Zhang, Min, Hall, Tracy ja Baddoo, Nathan: *Code Bad Smells: a review of current knowledge.* Journal of Software Maintenance and Evolution: Research and Practice, 23(3):179–202, 2011, ISSN 1532-0618.

[ZWN+06] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P. ja Vouk, M. A.: *On the value of static analysis for fault*

detection in software. IEEE Transactions on Software Enginee-
ring, 32(4):240–253, 2006, ISSN 0098-5589.

A ESLintin konfiguraatio

```
{
  "parser": "babel-eslint",
  "extends": "airbnb-base",
  "env": {
    "node": true,
    "mocha": true
  },
  "rules": {
    "semi": ["error", "always"],
    "comma-dangle": ["error", "always-multiline"],
    "max-len": [2, {"code": 200}],
    "no-plusplus": ["error", { "allowForLoopAfterthoughts": true }],
    "complexity": ["error", 8]
  }
}
```

Kohdeprojektin .eslintrc-tiedosto.

B SonarQuben konfiguraatio

```
# Projektin avain, joka on uniikki SonarQube-instanssissa
sonar.projectKey=project-backend

# SonarQuben käyttöliittymässä näytettävä projektin nimi ja versio
sonar.projectName=project-backend
sonar.projectVersion=1.0

sonar.sources=src

# Analyysiin sisällytettävät tiedostot
sonar.inclusions=src/**

# ESLint-pluginin konfiguraatio
sonar.eslint.eslintpath=node_modules/.bin/eslint
sonar.eslint.eslintconfigpath=.eslintrc
```

Kohdeprojektin sonar-project.properties-konfiguraatiotiedosto.

C Täydellinen listaus ESLintin raportoimista virheistä

Tässä liitteessä raportoidaan kaikki säännöt, joista ESLint havaitsi virheen. Taulukko alkaa seuraavalla sivulla. Säännön nimen lisäksi jokaisesta säännöstä kerrotaan kategoria johon se kuuluu, annetaan lyhyt kuvaus säännöstä sekä raportoidaan havaittujen virheiden lukumäärä.

Sääntö	**Kategoria**	**Kuvaus**	**Virheet lkm**
no-var	ECMAScript 6	Vaatii käyttämään muuttujien määrittelyssä avainsanan `var` sijaan `const` tai `let`.	204
comma-dangle	Mahdolliset virheet	Pakottaa yhdenmukaiseen pilkkujen käyttöön objektiliteraaleissa. Käytetty konfiguraatio vaatii pilkun objektin viimeisen elementin tai ominaisuuden jälkeen ainoastaan, mikäli sulkeva haka- tai aaltosulku on eri rivillä kuin objektin viimeinen elementti tai ominaisuus. Sääntö kieltää objektin viimeisen elementin tai ominaisuuden jälkeisen pilkun, mikäli elementti ja sulkeva haka- tai aaltosulku ovat samalla rivillä.	111
generator-star-spacing	ECMAScript 6	Määrittelee yhdenmukaisen tavan käyttää välilyöntejä *-merkin ympärillä generaattorifunktioissa. Käytetty konfiguraatio märittelee, että välilyönti tulee olla *-merkin jälkeen mutta ei ennen.	107
object-curly-spacing	Tyyli	Pakottaa yhdenmukaiseen välilyöntien käyttöön aaltosulkujen sisällä. Käytetyssä konfiguraatiossa pakotetaan aina käyttämään välilyöntejä aaltosulkujen sisällä, lukuunottamatta tyhjää lohkoa `"{}"`.	104
prefer-const	ECMAScript 6	Pakottaa määrittelemään muuttujat `let`-avainsanan sijaan `const`-avainsanalla, mikäli muuttujalle ei myöhemmin aseteta uutta arvoa.	57

Sääntö	**Kategoria**	**Kuvaus**	**Virheet lkm**
object-shorthand	ECMAScript 6	Pakottaa käyttämään lyhennettyä syntaksia	46
space-before-function-paren	Tyyli	Pakottaa yhtenäiseen välilyöntien käyttöön funktion sulkumerkin edessä. Käytetty konfiguraatio vaatii välilyönnin ennen sulkumerkkiä anonyymeissa funktioissa sekä async-nuolifunktioissa mutta ei salli sitä nimetyissä funktioissa.	46
vars-on-top	Hyvät käytänteet	Pakottaa pitämään kaikki muuttujien määrittelyt funktion näkyvyysalueen alussa.	41
strict	Strict mode	Sääntö sallii tai kieltää "strict"-tilan käytön. Käytetyssä konfiguraatiossa tilan käyttö on kielletty, sillä JavaScript-kääntäjä asettaa sen automaattisesti.	23
quote-props	Tyyli	Vaatii lainausmerkit objektiliteraalin ominaisuuden nimen ympärillä. Käytetty konfiguraatio vaatii lainausmerkit ainoastaan sellaisten literaalien kohdalla, joissa lainausmerkkien käyttämättömyys voisi johtaa epätoivottuihin tilanteisiin. Tällainen on esimerkiksi välilyönnin käyttö ominaisuuden nimessä. Lisäksi konfiguraatio ei vaadi lainausmerkkejä JavaScriptin avainsanojen eikä numeroiden käytettämisessä objektin ominaisuuden nimessä.	18

Sääntö	**Kategoria**	**Kuvaus**	**Virheet lkm**
keyword-spacing	Tyyli	Pakottaa yhdenmukaiseen välilyöntien käyttöön JavaScriptin avainsanojen kuten `function` ja `if` kanssa. Käytetyssä konfiguraatiossa välilyönti vaditaan avainsanaa ennen ja jälkeen. Lisäksi avainsanoille `case`, `return` ja `throw` sääntö on ylikirjoitettu niin, että välilyönti vaaditaan vain avainsanan jälkeen.	13
semi	Tyyli	Pakottaa yhdenmukaiseen puolipisteiden käyttöön. Käytetty konfiguraatio vaatii puolipisteen aina lausekkeen lopussa.	13
no-trailing-spaces	Tyyli	Kieltää ylimääräisten välilyöntien (whitespacen) käytön rivien lopussa.	12
prefer-arrow-callback	ECMAScript 6	Pakottaa käyttämään nuoli-funktiota (arrow function) takaisinkutsuina (callback) tai funktion argumenttina aina kun tämä on mahdollista. Käytetyssä konfiguraatiossa lisäksi kielletään nimettyjen funktioiden käyttäminen takaisinkutsuna tai funktion argumenttina sekä sallitaan `this`-avainsanan sisältyminen funktioilmaisuun, jota käytetään takaisunkutsuna tai funktion argumenttina kunhan kyseistä funktiota ei ole eksplisiittisesti sidottu (bound).	10

Sääntö	Kategoria	Kuvaus	Virheet lkm
spaced-comment	Tyyli	Pakottaa yhtenäiseen välilyöntien käyttöön kommentin alkumerkin "//"tai "/*"jälkeen sekä ennen loppumerkkiä "//"tai "*/". Käytetty konfiguraatio vaatii välilyönnin molemmissa edellämainituissa tilanteissa. Poikkeuksena ovat "+"ja ---merkit sekä "markereiksi"eli dokumentaatio-tyylisiksi tunnisteiksi määritellyt -"ja "!", jotka eivät vaadi välilyöntiä kommenttimerkkien yhteydessä.	10
quotes	Tyyli	Pakottaa yhtenäiseen joko yksin- tai kaksinkertaisten lainausmerkkien tai gravismerkin ("`", backquote) käyttöön. Käytetyssä konfiguraatiossa vaaditaan yksinkertaiset lainausmerkit aina kun niiden käyttö on mahdollista sekä sallitaan kaksinkertaiset lainausmerkit, mikäli merkkijonon sisällä on yksinkertaiset lainausmerkit, joille täytyisi muuten käyttää koodinvaihtomerkkiä (escape character).	7
no-multi-assign	Tyyli	Kieltää ketjutettujen sijoitusoperaatioiden käytön, esim. `a = b = c`.	6
indent	Tyyli	Pakottaa yhdenmukaiseen sisennystapaan. Käytetyssä konfiguraatiossa sisennys on määritelty kahden välilyönnin levyiseksi.	6
no-plusplus	Tyyli	Kieltää unaaristen operaattorien ++ ja -- käytön.Käytetyssä konfiguraatiossa kuitenkin sallitaan unaariset operaatiot for-loopin viimeisessä lausekkeessa.	6

Sääntö	**Kategoria**	**Kuvaus**	**Virheet lkm**
dot-notation	Hyvät käytänteet	Kehottaa käyttämään objektin ominaisuuksia käsiteltäessä piste-merkintätapaa hakasulkujen sijaan aina kun se on mahdollista.	5
max-len	Tyyli	Määrittelee rivin maksimileveyden. Tässä konfiguraatiossa maksimileveys on 200 merkkiä ja sisennysleveys kaksi merkkiä, eikä sääntö päde riveihin, jotka sisältävät URL:in, säännöllisen lausekkeen, yksin- tai kaksinkertaisilla lainausmerkeillä varustetun merkkijonon tai (template literal).	5
prefer-template	ECMAScript 6	Pakottaa käyttämään JavaScriptin template literaaleja merkkijonon ketjuttamisen sijaan.	5
no-multi-spaces	Hyvät käytänteet	Kieltää useamman kuin yhden välilyönnin käytön, mikäli kyseessä ei ole koodirivin sisennys.	3
block-spacing	Tyyli	Pakottaa yhdenmukaiseen välilyontien käyttöön yksirivisten koodilohkojen sisällä. Käytetty konfiguraatio vaatii välilyönnit sekä lohkon avaavan aaltosulun jälkeen että ennen sulkevaa aaltosulkua.	2
comma-spacing	Tyyli	Pakottaa yhdenmukaiseen välilyöntien käyttöön pilkkujen ympärillä muuttujien määrittelyn, taulukkojen, objektien, funktion parametrien ja sequencien yhteydessä. Käytetty konfiguraatio kieltää välilyönnin ennen pilkkua ja vaatii välilyönnin pilkun jälkeen.	2

Sääntö	**Kategoria**	**Kuvaus**	**Virheet lkm**
commplexity		TODO	2
key-spacing	Tyyli	Pakottaa yhtenäiseen välilyöntien käyttöön objekti-literaalien avaimen ja arvon välillä olevan kaksoispisteen ympärillä. Käytetyssä konfiguraatiossa välilyönti kielletään ennen kaksoispistettä ja vaaditaan kaksoispisteen jälkeen.	2
lines-around-directive	Tyyli	Vaatii tai kieltää tyhjien rivien käytön direktiivien ylä- tai alapuolella. Käytetyssä konfiguraatiossa tyhjä rivi vaaditaan sekä direktiiviä ennen että jälkeen.	2
no-useless-concat	Hyvät käytänteet	Kieltää tarpeettomat merkkijonojen ketjutukset. Esimerkiksi `var foo = "a" + "b";` tulee kirjoittaa muodossa `var foo = "ab";`.	2
require-yield	ECMAScript 6	Ei salli generaattori-funktioita, joilla ei ole JavaScriptin `yield`-avainsanaa.	2
space-before-blocks	Tyyli	Pakottaa yhdenmukaiseen välilyöntien käyttöön ennen lohkoa avaavaa aaltosulkua. Sääntö ei kuitenkaan ota huomioon ->-merkkiä ja avainsanoja edeltävistä välilyönneistä, sillä näille on määritelty omat säännöt (arrow-spacing ja keyword-spacing).	2
consistent-return	Hyvät käytänteet	Vaatii, että funktion jokainen haarauma joko palauttaa määritellyn arvon `return`-ilmaisussa, tai arvoa ei määritellä missään poluista ja palautetaan pelkkä tyhjä `return`-ilmaisu.	1

Sääntö	Kategoria	Kuvaus	Virheet lkm
eqeqeq	Hyvät käytänteet	Vaatii käyttämään === tai !== sen sijaan, että käytetään == tai !=. Käytetty konfiguraatio ei ota säännössä huomioon tyhjään (null) literaaliin vertaamista.	1
no-else-return	Hyvät käytänteet	Kieltää if-lauseessa else-haaran turhan käytön. Mikäli if-lohkossa palautetaan jotakin return-avainsanalla, else-haaraa ei tarvita, vaan sen siltö voidaan sijoittaa if-lauseen ulkopuolelle.	1
no-unneeded-ternary	Tyyli	Kieltää kolmiarvoisten operaattorien käytön, mikäli yksinkertaisempi vaihtoehto on olemassa.	1
padded-blocks	Tyyli	Pakottaa yhtenäiseen tyhjien rivien käyttöön koodilohkojen alussa ja lopussa. Käytetyssä konfiguraatiossa kielletään tyhjät rivit luokkien ja koodilohkojen alussa ja lopussa.	1
space-infix-ops	Tyyli	Pakottaa käyttämään välilyöntejä infix-operaattorien ympärillä.	1

Taulukko C.1: ESLintin havaitsemat virheet